Smart Data Accelerator Interface ("SDXI") Specification

Version 1.0

ABSTRACT: Smart Data Accelerator Interface (SDXI) is a standard for a memory-to-memory data mover and acceleration interface.

This document has been released and approved by the SNIA. The SNIA believes that the ideas, methodologies and technologies described in this document accurately represent the SNIA goals and are appropriate for widespread distribution. Suggestions for revisions should be directed to https://www.snia.org/feedback/

SNIA Standard

November 28, 2022

USAGE

DISCLAIMER

The information contained in this publication is subject to change without notice. The SNIA makes no warranty of any kind with regard to this specification, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The SNIA shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this specification.

Revision History

Revision	Date	Comments
1.0	November 28, 2022	Official Release
0.9.7-rev0	November 20, 2022	Approved by SNIA membership
0.9.7	September 5, 2022	Public Review
0.9	July 2021	First Public Preview
0.7	September 2020	Starting point Draft Contribution to SNIA SDXI TWG

Suggestions for revisions should be directed to https://www.snia.org/feedback/.

SDXI TWG 1.0 Contributors

Curtis Ballard, HPE

Beau Beachamp, MemVerge

Richard Brunner, VMware

Xiangping Chen, Dell

Don Dutile, IBM

Paul Hartke, AMD

Shyam Iyer, Dell Inc

Travis Hamilton, Arm

Brian Hirano, Micron

Frederick Knight, NetApp

Santosh Kumar, SK Hynix

James Leighton, Western Digital

Bill Martin, Samsung

John Maroney, Micron

J Metz, AMD

William Moyes, AMD

Philip Ng, AMD

Murali Ravirala, Microsoft

Dwight Riley, HPE

Alexandre Romana, Arm

Glen Sescila, Dell Inc

Paul Von Stamwitz, Fujitsu

Jason Wohlgemuth, Microsoft

Table of Contents

1 SDXI: Overview

This document describes SDXI, an architectural data-mover device for server platforms. SDXI aims to provide a variety of benefits including architectural stability, modularity, virtualization-friendly programming interface, both user and kernel mode support, and new capabilities designed to accelerate virtualized workloads.

1.1 Scope

1.2 Documentation Conventions

This specification adheres to various conventions found in the "IEEE Standards Board Operations Manual – Clause 6" at "https://standards.ieee.org/about/policies/opman/sect6/". A few are described here.

1.2.1 Shall, Should, May, and Can

- The word "shall" is used to indicate mandatory requirements strictly to be followed in order to conform to the Specification and from which no deviation is permitted ("shall" equals "is required to").
- The use of the word "must" is deprecated and shall not be used when stating mandatory requirements; "must" is used only to describe unavoidable situations.
- The use of the word "will" is deprecated and shall not be used when stating mandatory requirements; will is only used in statements of fact.
- The word "should" is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (should equals is recommended that).
- The word "may" is used to indicate a course of action permissible within the limits of the Specification (may equals is permitted).
- The word "can" is used for statements of possibility and capability, whether material, physical, or causal ("can" equals "is able to").

1.2.2 Normative vs. Informative

Normative material is information required to implement the standard and is therefore officially part of the standard. Informative material is provided for information only and is therefore not officially part of the standard.

All sections in this document are normative, unless they are explicitly indicated to be informative at the beginning of each section.

1.2.3 Reserved

The following applies to the term "Reserved", "rsvd", and "rsv":

- The contents, state, or information are not specified at this time.
- Any field, feature, capability, etc. marked "Reserved", "rsvd", or "rsv" is subject to change.
- We may use these terms interchangeably.

1.2.4 Developer Notes

Developer Notes do not specify normative or optional requirements. They are included for clarification and illustration only. These notes are delineated by:

> Developer Note: This is such a note …

1.2.5 Abbreviations and Terminology

Term	Definition
Administrative Context	A context that supports the use of administrative operations used for managing SDXI Functions and their contexts.
AKey	Access Key. See *"3.2.5, Access Key (AKey) Table Entry (AKEY_ENT)"*.
ATS	Address Translation Services. Defined in the PCI Express Base Specification.
Context	A Context refers to a descriptor ring used for executing operations, along with all associated memory data structures such as control and status information
Cross-Function	An operation where the SDXI Function owning the target resource differs from the requesting SDXI Function.
DMA	Direct Memory Access
DSH	Data Steering Hint See *"3.6, Data Steering Hints (DSH)"*.
Function Group	A collection of SDXI functions that may access each other's memory and interrupt resources.
GPA	Guest Physical Address
GVA	Guest Virtual Address
HPA	Host Physical Address
HVA	Host Virtual Address
IO	Input Output
IOMMU	IO Memory Management Unit

Term	Definition
Local	Local or Local Space (referenced in various structures by an "sfunc" field of "0") corresponds to the SDXI function hosting the context and descriptor.
MMIO	Memory mapped IO
MMU	Memory Management Unit
P2V	An administrative command from a Physical Function that targets a Virtual Function.
PASID	Process Address Space Identifier. The combination of PASID and Requester ID identifies the address space used by a transaction on the PCIe bus.
PRI	Page Request Interface. Defined in the PCI Express Base Specification.
Remote	May be used by an SDXI Function owning a target resource to refer to a separate requesting SDXI Function. May also be used by a requesting SDXI Function to refer to a target resource owned by a different SDXI Function. Also see Cross-Function.
Requester ID	The PCIe bus, device and function number used by PCIe SDXI Functions as part of DMA, interrupt and address translation operations.
RKey	Receiver Access Key. See *"3.3.2, Receiver Access Key (RKey) Table"*.
SDXI Function	A device that implements the SDXI specification. This may be either a physical or virtual function.
sfunc	An opaque SDXI Function identifier that maps uniquely to an SDXI function. For example, in a PCIe implementation of SDXI, "sfunc" could map to a PCIe Requester ID. This is used when accessing AKey-referenced data structures. The value of "0" is defined to reference the local hosting SDXI function; all other encodings refer to other non-local SDXI functions.

1.2.6 Other Clarifying Notes

1.2.6.1 Index

Index as used in this specification refers to a table index to a table-specific entry. In order to address the corresponding table entry in memory, the index is multiplied by the size of the table-specific entry and added to the beginning address of the table. For example, if a table starting at address 0x1000 has entries that are 64 bytes in size, then an index into that table needs to be multiplied by 64 and added to 0x1000 to determine the correct memory location of the entry.

1.3 References

- PCI Express Base Specification, Revision 5.0

1.4 Document Constants

These are defined solely for tooling scripts related to the document.

Table 1–1: Document CONSTANTS[^0]

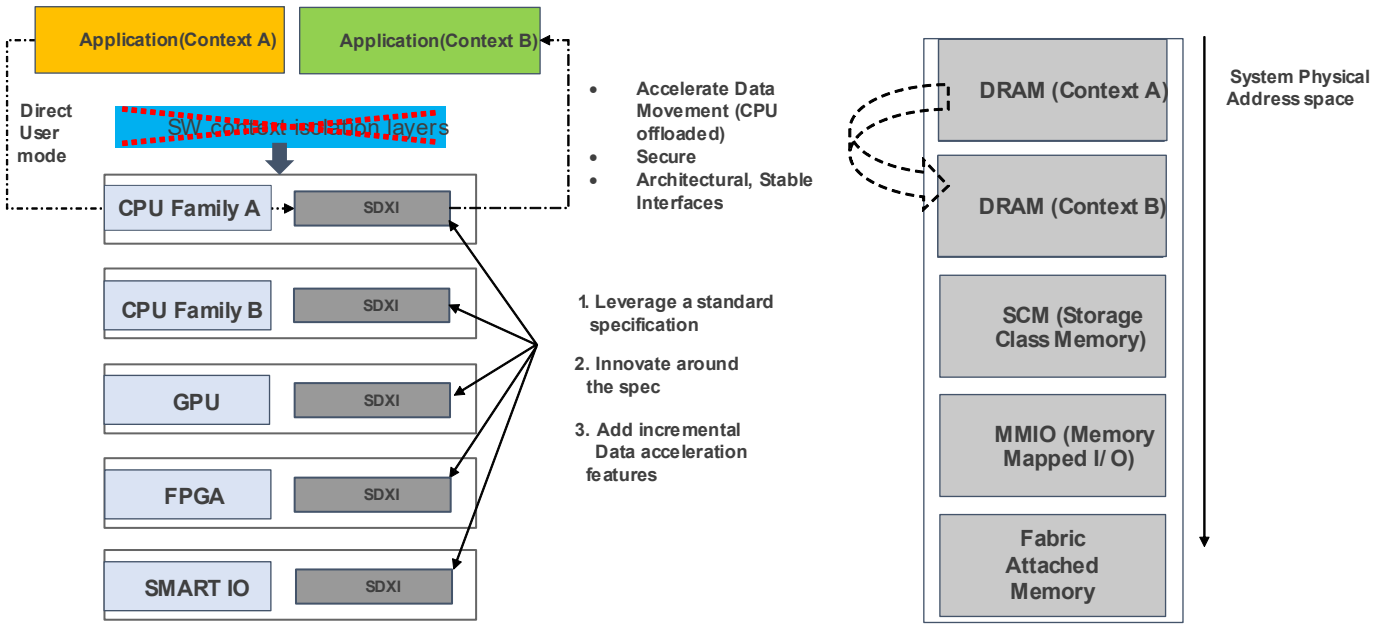
Constant	Value	Description
DOC_REVISION	1.0	Document Revision. Must be entered manually here.

2 Background

(The entirety of this chapter is informative.)

The concept of a Direct Memory Access (DMA) data mover device to offload software-based copy loops is well-known. Such offloading is desirable to free up CPU execution cycles. But adoption has been mostly limited to specific privileged software and I/O use cases employing very device-specific interfaces that are not forward compatible. The current limitations make user-mode application usage challenging in a non-virtualized environment and nearly impossible in a multi-tenant virtualized environment. The figure below shows a vision for such an architectural data mover.

Figure 2-1: Architectural Data Mover



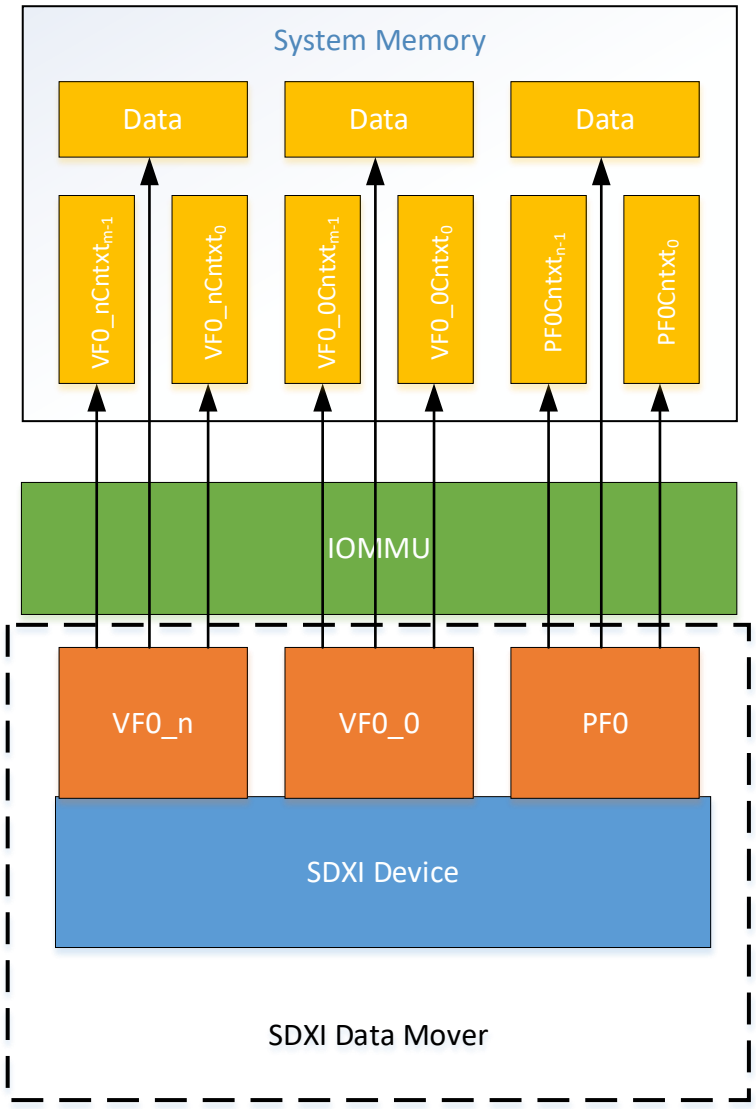
We have developed the SDXI Platform Data Mover (aka "SDXI") to provide an architectural interface to address these limitations:

- An extensible, forward-compatible interface that is independent of actual data mover implementations and underlying I/O interconnect technology.
- A standard interface for user-mode address-space to address-space data movement without the need of mediation by privileged software once a connection has been established.
- A standard interface for privileged software to control the data mover including connection management and data movement between multiple address spaces
- An interface that can be abstracted or virtualized by privileged software to allow greater compatibility of workloads or virtual machines across different servers.
- A well-defined capability to quiesce, suspend, and resume the architectural state of a per-address-space data mover to allow "live" workload or virtual machine migration between servers.
- A forward compatible interface that ensures pre-existing software drivers including user-mode drivers can operate on future hardware implementations without modification.
- The ability to incorporate additional offloads in the future without modification to the architectural interface.

2.1 Architected Platform Data Mover

The basic SDXI architecture consists of some number of Smart Data accelerators that are enumerated as one or more SDXI functions. Each function may support 1 or more contexts with each context having a single descriptor ring. SR-IOV is optionally supported.

Figure 2-2: Basic SDXI Architecture



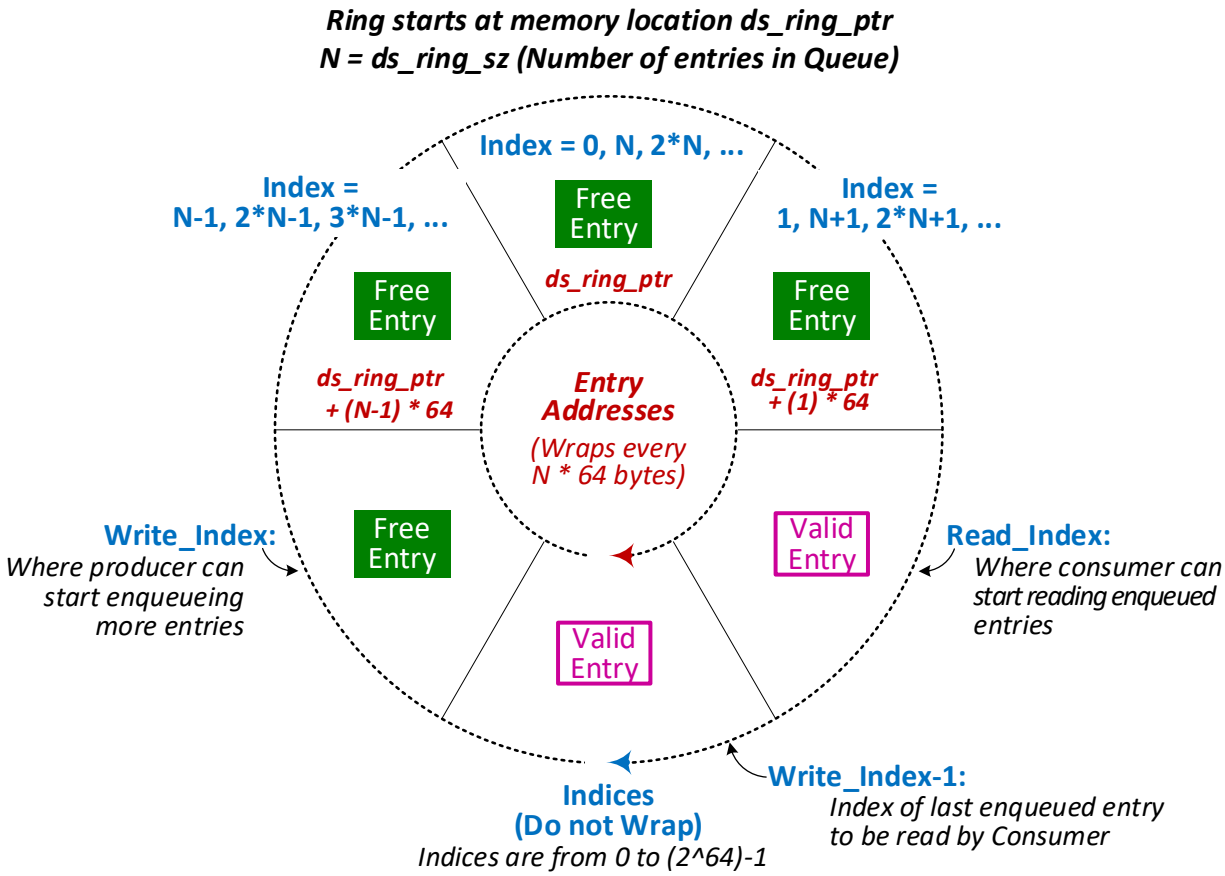
2.1.1 SDXI Descriptor Ring

An SDXI descriptor is a naturally aligned 64-byte entry that instructs an SDXI function to perform a given operation. Descriptors are placed in a circular ring that starts at a specified address (ds_ring_ptr). The ring is contiguous at the translation level configured for the SDXI function. The ring is configured to contain a given number of descriptor ring entries (ds_ring_sz). The number of bytes allocated in memory for the ring is: (ds_ring_sz * 64).

A set of related operations form an operations group. An operation requires a particular descriptor format to indicate parameters for the operation.

The ring and all of its related system memory data structures comprise a "context". SDXI uses a 2-level hierarchy of context tables (Context Table Level 2, Context Table Level 1) to enumerate the components of the context. The concatenation of the offsets used to enumerate a context in both Context tables yields the 16-bit "context_number" that is associated with the context. (Note that the Context Tables point to a context but are not themselves part of the context.)

Figure 2-3: SDXI Descriptor Ring



EntryAddress = ds_ring_ptr + ((Index % ds_ring_sz) << 6)
Write_Index – Read_Index <= ds_ring_sz

A circular ring requires "start" and "end" indicators. Rather than using memory pointers to track these, an SDXI descriptor ring uses 64-bit *unsigned* logical indices to indicate the start (Read_Index) and end (Write_Index) of the descriptor ring. This simplifies various calculations for SW and HW alike. The logical

indices need only be mapped onto descriptor ring addresses when writing or reading a ring entry at a given Index. All rings use the same mechanism to communicate between producer and consumer.

From the perspective of software, SDXI has two kinds of descriptor rings:

- A software producer ring. Software writes descriptors on to the ring, increments Write_Index, and writes to the doorbell to signal the SDXI function that a new descriptor has been written. The SDXI function reads from the ring, increments Read_Index, and performs the requested operation. With a few exceptions noted in the next list item, all rings are software producer rings.
- A software consumer ring. The SDXI function writes log messages (using the format of descriptors) on to the ring, increments Write_Index, and can be configured to generate an interrupt to signal that a new message has been written. Software reads from the log ring, increments Read_Index, and processes the requested message. The Error Log is an example of this kind of ring.

2.1.2 Virtualization Support

SDXI is architected specifically to allow operation within a virtualized environment.

Live migration is supported through a variety of accommodations in the programming model. During live migration, a real hardware SDXI implementation may be stopped by the Hypervisor. Once stopped, there is no hidden state. All critical state is located in either MMIO registers or in system memory. Another instance may be configured with this state and restarted. It is possible to arbitrarily transition between hardware and software emulated implementations.

The programming model allows a Hypervisor to hide features from newer implementations and specifications from a VM. This facilitates migration of a VM built around older versions of the specification onto newer hardware.

Specific use cases around virtualized environments may be accelerated by SDXI. See *"2.3, Address Space Control Examples".*

An assignable entity is a portion of an SDXI device which may be assigned to a VM. An individual SDXI Function, regardless of its device model, represents an assignable entity. For example, in a PCIe SR-IOV implementation of SDXI, each Physical Function (PF) and Virtual Function (VF) would represent separate SDXI Functions. In general, it is assumed that a Hypervisor will control SDXI PFs and allow SDXI VFs to be directly assigned into VMs.

2.2 DMA Addressing Modes

SDXI Functions generate DMA requests with address space identifiers that allow a platform-specific IOMMU to map the associated addresses. Depending on the overall configuration of the SDXI Function and the IOMMU, DMA addresses may be treated as Guest Virtual, Guest Physical or Host Physical for translation purposes.

SDXI provides granular control over the address space identifier used when accessing various SDXI memory data structures (ex. descriptor ring) and each data buffer. These controls allow a single SDXI operation to reference multiple address spaces. For example, an SDXI copy operation may copy data from one address space to another.

DMA requests may be generated with a PASID value, or without PASID as a part of the address space identifier. Additionally, SDXI utilizes the SDXI Function identifier as part of the address space identifier. For example, an SDXI Function constructed as a PCIe SR-IOV VF would use the VF's PCIe Requester ID as the Function identifier when making DMA requests to data structures owned by the VF.

2.2.1 Address Space Identifier Control

SDXI provides control over the address space identifier used when accessing various control structures in memory. Specifically, the PASID portion is controlled, depending on the structure, through either an mmio register or a memory data structure (Context Level 1 Table Entry). The SDXI Function identifier used when accessing control structures is always that of the Function owning the control structures and executing the descriptor.

SDXI determines the address space of each data buffer using an AKey value provided in the descriptor, which is used to look up an AKey table entry in memory. An AKey table entry contains PASID controls which help form the address space identifier used to access the data buffer. Additionally, an AKey table entry contains a tgt_sfunc value which also helps form the data buffer's address space identifier. The sfunc value is an opaque handle used to abstract the SDXI Function identifier.

Use of the tgt_sfunc mechanism to access a data buffer owned by a different SDXI Function than the one executing the descriptor is referred to as a Cross-Function operation. Two or more SDXI Functions are involved. This mechanism may be used, for example, to perform a transfer of data from one VM to another.

Cross-Function operations utilize an additional RKey mechanism to authorize the data buffer access. (See *"3.3, SDXI Cross-Function Access"* for more details).

2.2.2 SDXI Function Groups

Cross-Function operations are only supported between SDXI Functions that reside within the same SDXI function group. An SDXI PF and its associated VFs always belong to the same function group. A function group may contain multiple PFs and VFs. A system may contain multiple disconnected function groups.

Function groups are discovered using the MMIO_GRP_ENUM register (see "*3.3.1, SDXI Function Group*" for details). The method of communication between SDXI functions within a function group is outside the scope of this specification.

2.2.3 Unpinned Memory Access

SDXI supports the use of unpinned memory. For example this can be accomplished in an PCIe implementation through the use of the PCIe-defined ATS and PRI features. The SDXI programming model does not explicitly identify whether data structures reside in unpinned memory. If the PCIe ATS and PRI features are enabled within a function, an SDXI implementation must assume that any memory accesses associated with that function's address spaces may target unpinned memory.

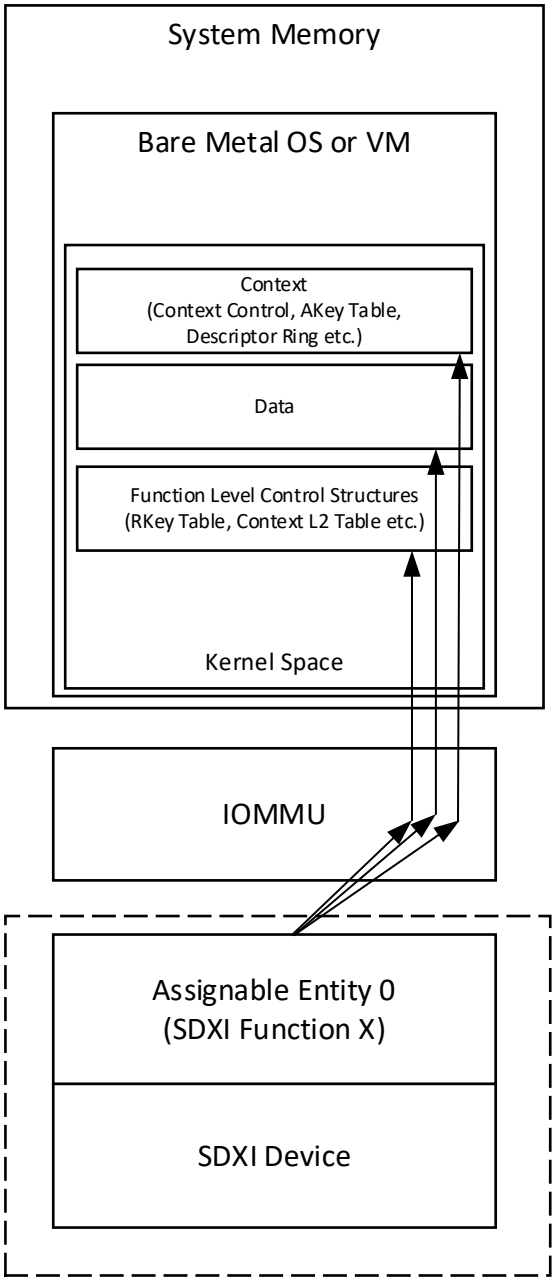
2.3 Address Space Control Examples

The address space identifier controls in SDXI enable a wide range of capabilities for accessing data in memory. The following sub-sections describe several example usage models.

2.3.1 Single-Function, Single Address Space Example

In this example, a single SDXI Function is utilized. All control and data structures are located in the same address space. This may, for example, represent a case where SDXI is controlled by a kernel driver and is accessing kernel data structures within a VM or a bare metal OS.

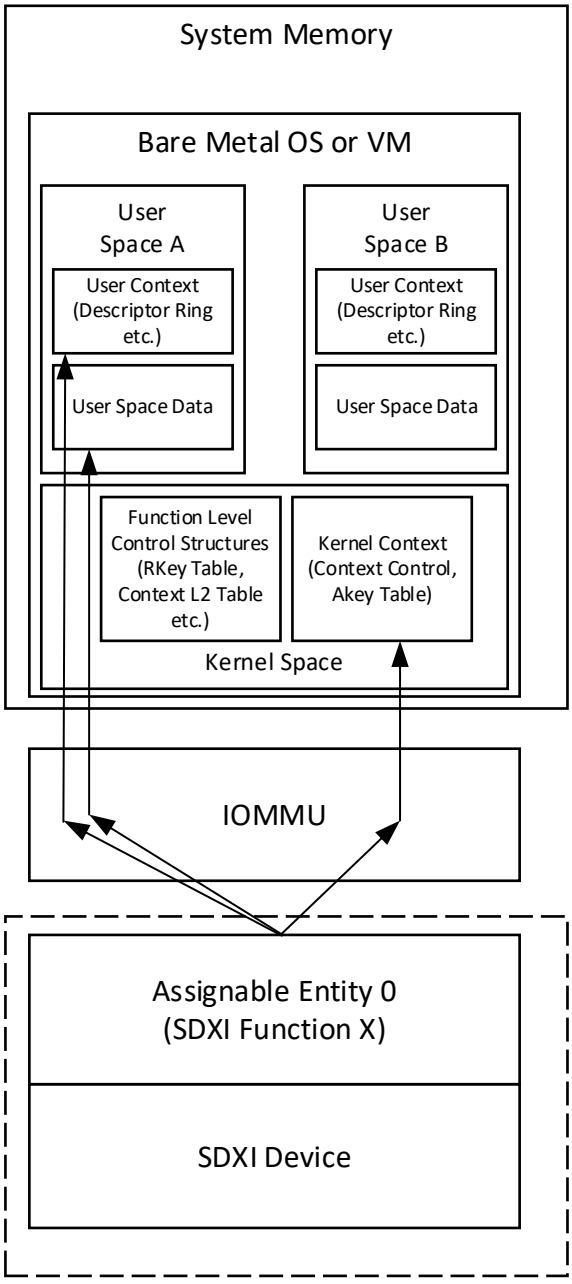
Figure 2-4 Single Address Space Example



2.3.2 Single-Function, User Mode Access Example

In this example, control of the SDXI Function is split between a kernel mode driver and a user mode driver. The kernel mode driver manages function level control structures and a portion of each SDXI in kernel memory. The user mode driver manages descriptors in user space memory and accesses data buffers in the same user space.

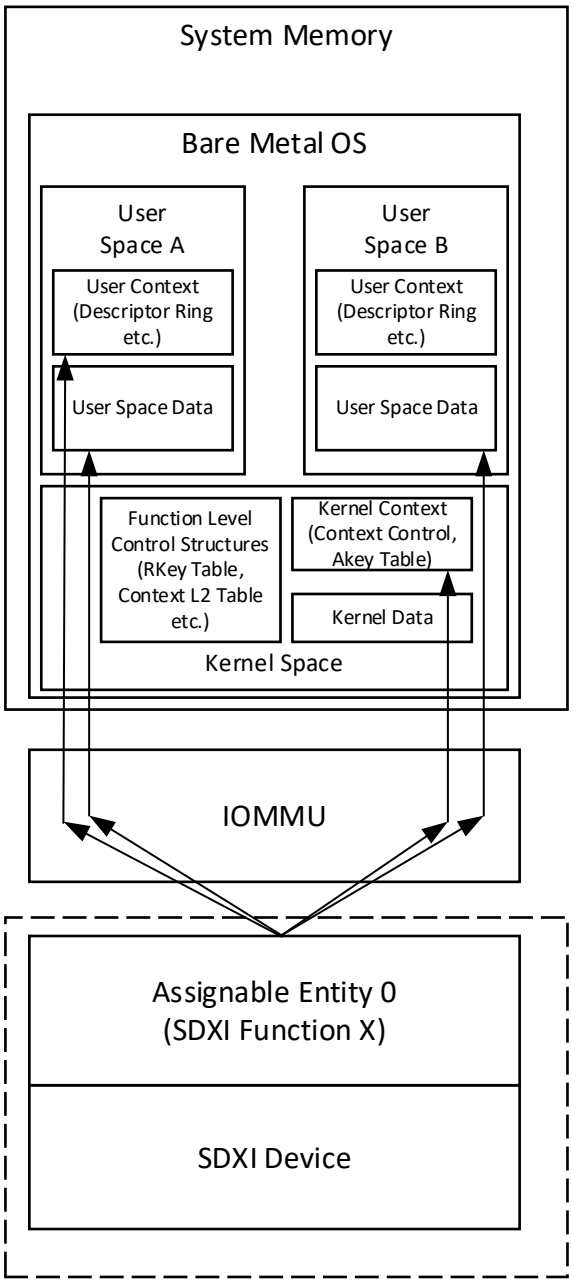
Figure 2-5 User Mode Access Example



2.3.3 Single-Function, Multiple Address Space Example

Building on the prior example, this case moves one of the data buffers into a different address space. This may be used, for example, to copy data between different user spaces or between kernel and user space.

Figure 2-6 Multiple Address Space Transfer Example



2.3.4 Cross-Function Transfer Example

In this example there are multiple SDXI Functions. SDXI Function Cntl is mapped to VM0, SDXI Function Src is mapped to VM1 and SDXI Function Dst is mapped to VM2. SDXI Function Cntl is executing descriptors from a user process in VM0 and is instructed to perform a Cross-Function copy of data using Function Src to access the source buffer and Function Dst to access the destination buffer. Additionally, the address space controls point the source buffer to a user process in VM1 and the destination buffer to a user process in VM2. As with the prior examples, data buffers may be located in kernel or user space within each of the VMs. "*Figure 2-4 Single Address Space Example*" shows the example data flow where all of the SDXI functions reside within the same SDXI device. "*Figure 2-5 User Mode Access Example*" shows a similar example where the SDXI functions reside within 2 different SDXI devices.

Figure 2-4 Cross-Function Transfer Example Within Single SDXI Device

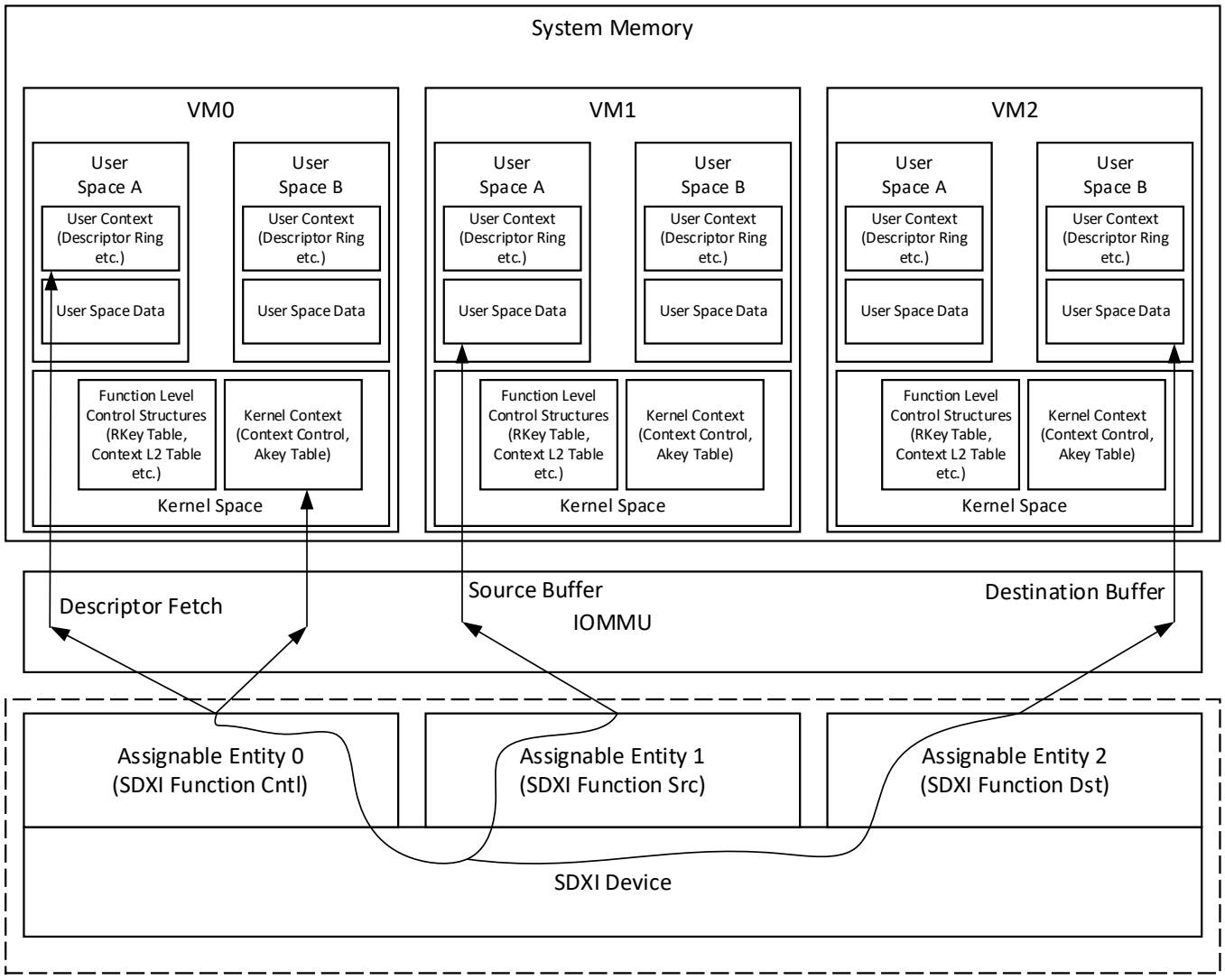
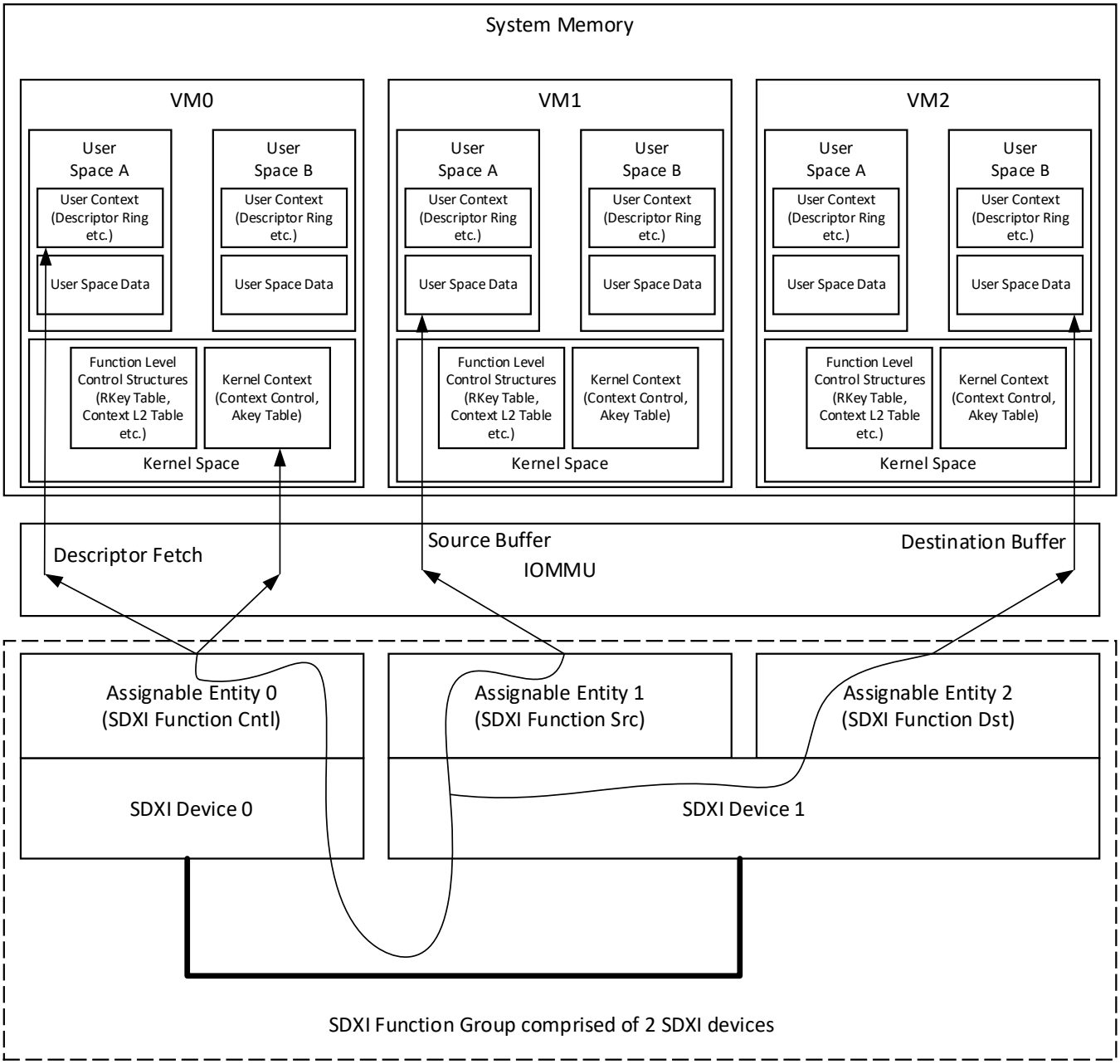


Figure 2-5 Cross-Function Transfer Between SDXI Devices Example



2.4 Modularity and Expandability

SDXI is architected as a modular and expandable framework for offload functions. While the initial specification is focused on copy operations, additional offloads may be defined and added to newer versions of the specification, as well as newer implementations. All new features shall be explicitly discovered and enabled by supporting software. Implementations may choose to implement different sets of offload capabilities.

In general, all SDXI OS/VM and user-level software is expected to operate transparently on any hardware implementation that supports the same or newer version of the specification that the software was designed for, as long as all of the expected offload capabilities are present.

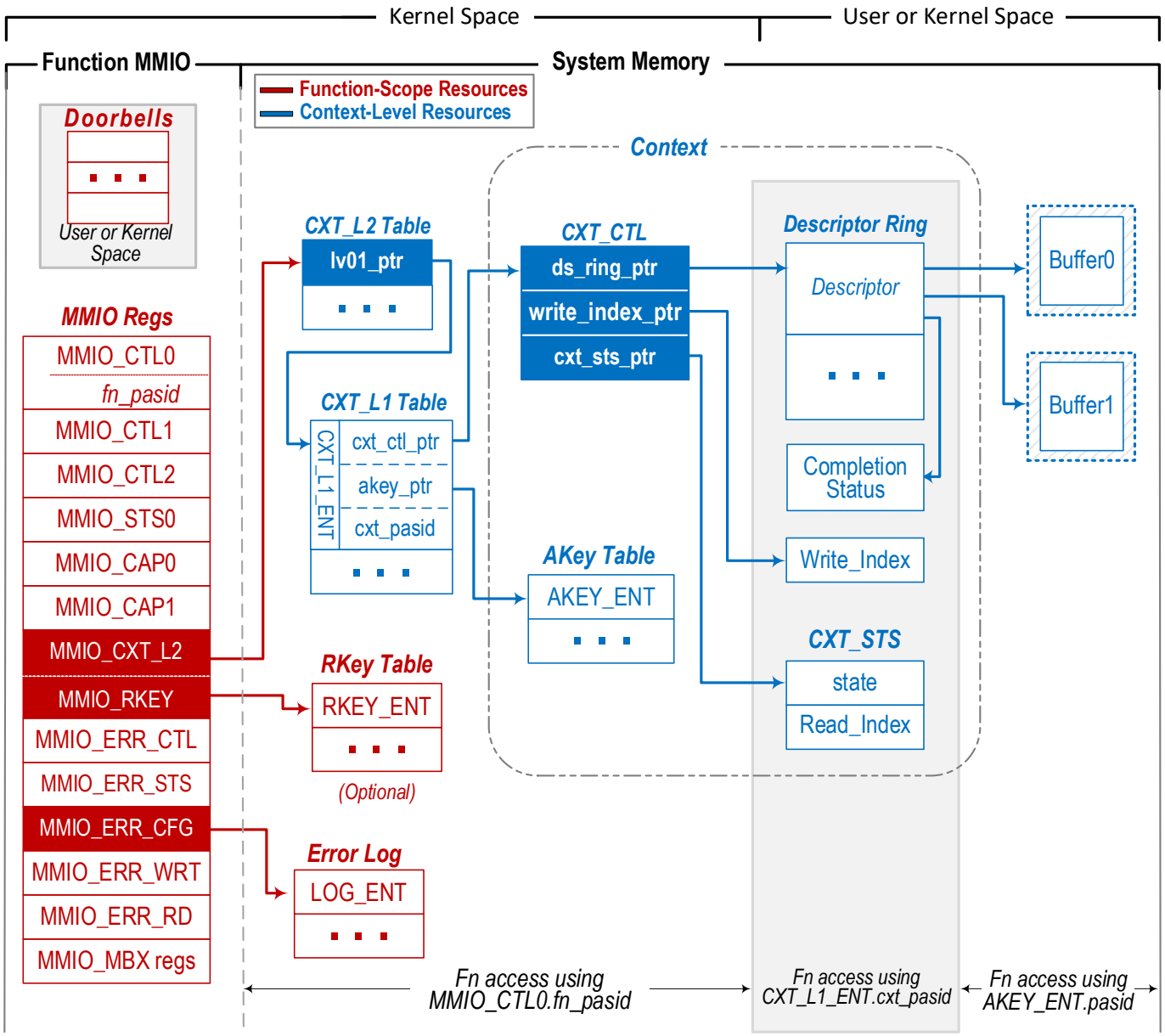
2.5 Endian Format Support

The SDXI specification is written assuming a little-endian architecture. Support for other endian formats is outside the scope of this specification.

3.1 Overview

The SDXI architecture is independent of the underlying I/O interconnect. However, the SDXI specification includes a PCIe binding and PCIe is frequently used for examples throughout this specification. The following figure depicts all of the memory-addressed data structures used by an SDXI function. In order to facilitate efficient software-based virtualization, the majority of an SDXI function's architectural state resides in system memory and is described in this chapter. The remaining state resides in a small number of MMIO control registers (described in *"9, MMIO Control Registers"*) and PCI Function configuration space registers (described in *"8, SDXI PCI-Express Device Architecture"*). This layout of memory structures is designed to facilitate selective trapping by privileged software.

Figure 3-1: Memory-Addressed Data Structures



Software shall ensure that the SDXI function has the required read and write access for each memory structure as shown in *"Table 3–1: Memory Structure* Summary*"*.

Table 3–1: Memory Structure Summary

Structure	Req"d SDXI FN Access	Alignment	Maximum Structure Size	Entry Size	PASID Used
CXT_L2 Table	R	4 KByte	**4 KBytes**	8 Bytes	Derived from MMIO_CTL0. If fn_pasid_vl == 1, then fn_pasid is used.
CXT_L1 Table	R	4 KByte	**4 KBytes**	32 Bytes	
CXT_CTL	R	64 Byte	64 Bytes	n/a	
Error Log	W	4-KByte	Let sz = MMIO_CAP1.max_errlog_sz; 0 <= sz <= 9; size = 2**(23 + sz) max_size = **2**32 byte**s.	64 Bytes	
AKey Table	R	4 KByte	Let sz = MMIO_CAP1.max_akey_sz; 0 <= sz <= 8; size = 2**(12 + sz) max_size = **2**20 bytes**.	16 Bytes	
RKey Table	R	4 KByte	Let sz = MMIO_RKEY.tbl_sz; 0 <= sz <= 8; size = 2**(12 + sz) max_size = **2**20 bytes**.	16 Bytes	
Descriptor Ring	R/W	64 Byte	Let sz = MMIO_CAP0.max_ds_ring_sz; 0 <= sz <= 22; size = 2**(16 + sz) max_size = **2**38 bytes**.	64 Bytes	Derived from CXT_L1_ENT. If pv == "1", then cxt_pasid is used.
CXT_STS	R/W	16 Byte	**16 Bytes**	n/a	
(CXT_STS.) Read_Index	R/W	8 Byte	**8 Bytes**	n/a	
Write_Index	R	8 Byte	**8 Bytes**	n/a	
Completion Status Block (CST_BLK)	R/W	32-Byte	**32 Bytes**	n/a	
Atomic Return Data	R/W	Operand size is 4 or 8 bytes. Naturally aligned.		n/a	
Data Buffer	R/W as req"d	n/a	Let sz = MMIO_CAP1.max_buffer; 0 <= sz <= 11; size = 2**(21 + sz) max_size = **2**32 bytes**.	n/a	Derived from AKEY_ENT. If pv == 1 AND tgt_sfunc == 0, then pasid is used.

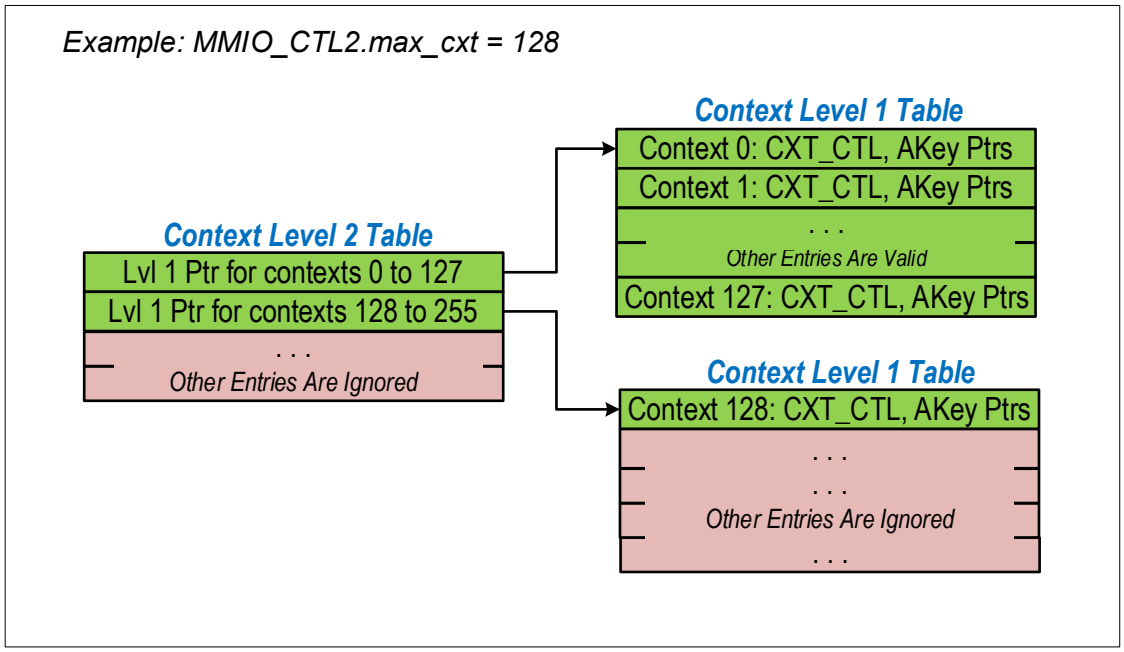
3.2 Context Data Structures

An SDXI context represents the memory structures needed to directly monitor or control the operation of a descriptor ring. An SDXI context consists of a descriptor ring, and the associated Context Control (CXT_CTL), Context Status (CXT_STS), AKey Table, and Write_Index structures. Memory buffers, completion signals, and atomic results which descriptor ring entries operate upon are not considered part of the SDXI function context; however, they are an important part of the additional corresponding state that software manages directly in order to use the descriptor ring.

SDXI uses a 2-level hierarchy of context tables (Context Table Level 2, Context Table Level 1) to enumerate the components of the context. The concatenation of the 9-bit Level 2 offset with the 7-bit Level 1 offset yields the 16-bit "context_number" that is associated with the context. (Note that the Context Tables point to a context but are not themselves part of the context.) MMIO_CAP1.max_cxt indicates the maximum context number supported by the function. Software may further reduce the available contexts by programming a smaller value into MMIO_CTL2.max_cxt.

An SDXI function shall not access portions of the context tables associated with context numbers greater than MMIO_CTL2.max_cxt. See the example below.

Figure 3-2: MMIO_CTL2.max_cxt Example



The SDXI specification uses 64-bit pointers. When data structures are required to be aligned to a certain size, the lower bits of the pointer may be used for other purposes. This may result in the pointer field of a data structure or register being less than 64-bits.

3.2.1 Context Level 2 Table

The Context Base Table register points to the 4KB level 2 table containing an array of level 2 table entries described below. Each of those entries is a pointer to a level 1 table. A level 2 table contains 512 level 1 pointers.

Figure 3-3: Context Level 2 Table Entry (CXT_L2_ENT)

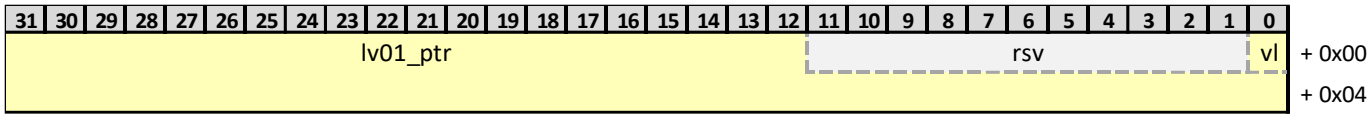


Table 3–2: Context Level 2 Table Entry (CXT_L2_ENT[^3])

Field	Bits	Subfield	Description
u64 lv01_ptr;	000	vl	Valid. When 1, indicates the other bits in this data structure are valid. When 0, all other bits in this data structure shall be ignored.
	011:001	rsvd	Shall be set to zero.
	063:012	lv01_ptr	Pointer to the start of a Context Level 1 table. This points to an aligned 4K region of memory.

3.2.2 Context Level 1 Table

Each of the level 1 table structures is a 4K naturally aligned piece of memory containing an array of context level 1 table entries described below. A Context level 1 table has 128 entries.

Figure 3-4: Context Level 1 Table Entry (CXT_L1_ENT)

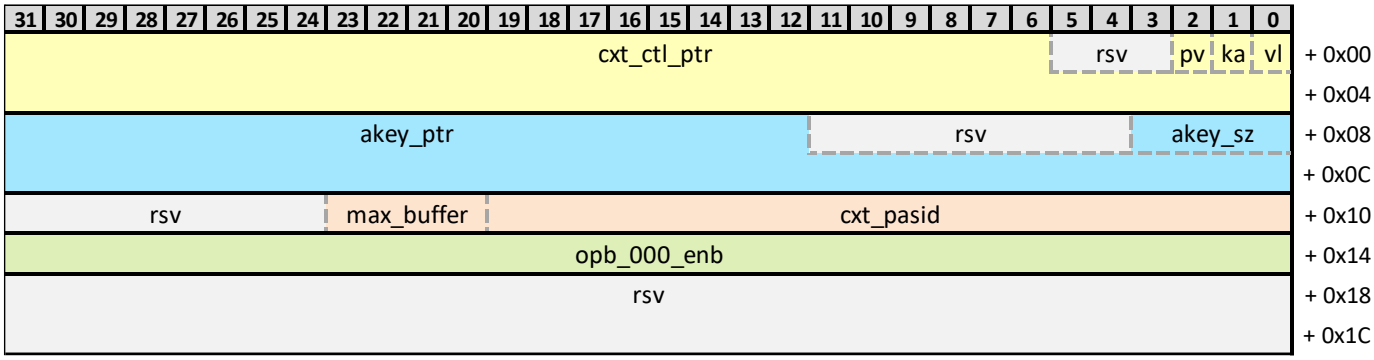


Table 3–3: Context Level 1 Table Entry (CXT_L1_ENT[^3])

Field	Bits	Subfield	Description
u64 cxt_ctl_ptr;	000	vl	Valid. When 1, indicates the other bits in this data structure are valid. When 0, all other bits in this data structure shall be ignored.
	001	ka	Keep-Active-Hint. Hint to the SDXI function. Does not affect context state. When "1", the function operates the context normally and should execute new descriptors. This should be set to "1" by privileged software when it starts the context. When "0", the function may choose to stop executing new descriptors in anticipation that software will soon stop the context. This may reduce the time that software must wait for stopping the context later.
	002	pv	PASID Valid. Indicates the cxt_pasid field is valid.
	005:003	rsvd	Shall be set to zero.
	063:006	cxt_ctl_ptr	Pointer to the Context Control (CXT_CTL). This points to a 64B aligned region of memory.
u64 akey_ptr;	067:064	akey_sz	AKey Table Size. Indicates the size of the AKey table referenced by akey_ptr. The table size is encoded as 2**(akey_sz+12) bytes or 2**(akey_sz+8) entries. Encodings greater than 0x8 are reserved.
	075:068	rsvd	Shall be set to zero.
	127:076	akey_ptr	Pointer to the start of an AKey table. This points to a 4Kbyte aligned region of memory.
u32 misc0;	147:128	cxt_pasid	If pv=1, indicates the PASID value used to access the Write_Index, CXT_STS, CST_BLK, Atomic Return Data, and Descriptor Ring. If pv=0, this field is not used and the Write_Index, CXT_STS, CST_BLK and Descriptor Ring are accessed without a PASID.
	151:148	max_buffer	Indicates the maximum data buffer size supported by this context. The size is encoded as 2**(max_buffer+21). Descriptor type or the size of the buffer length field within certain types of descriptors may further limit the maximum data buffer size for certain types of operations. This field should not be set to exceed MMIO_CTL2.max_buffer
	159:152	rsvd	Shall be set to zero.
u32 opb_000_enb;	191:160		Each bit in this field, when set to "1", indicates that a certain descriptor operation group is enabled within this context; when "0", it is not. The encoding of the bits matches the MMIO_CAP1.opb_000_cap register. See *"5.1, Descriptor Operations"* and *"Chapter 6, SDXI Descriptor and Operation Specification"* for more details.
u8 rsvd_0[8];	255:192		Shall be set to zero.

3.2.3 Context Control (CXT_CTL)

The Context Control contains control information for a single descriptor ring.

Software shall expose the memory containing the Context Control as readable to the SDXI function hosting the context. Refer to *"Table 3–1: Memory Structure* Summary*"* for further requirements on how software shall expose the structures pointed to by CXT_CTL to the SDXI function. This structure is intended to be controlled by privileged software.

Figure 3-5: Context Control (CXT_CTL)

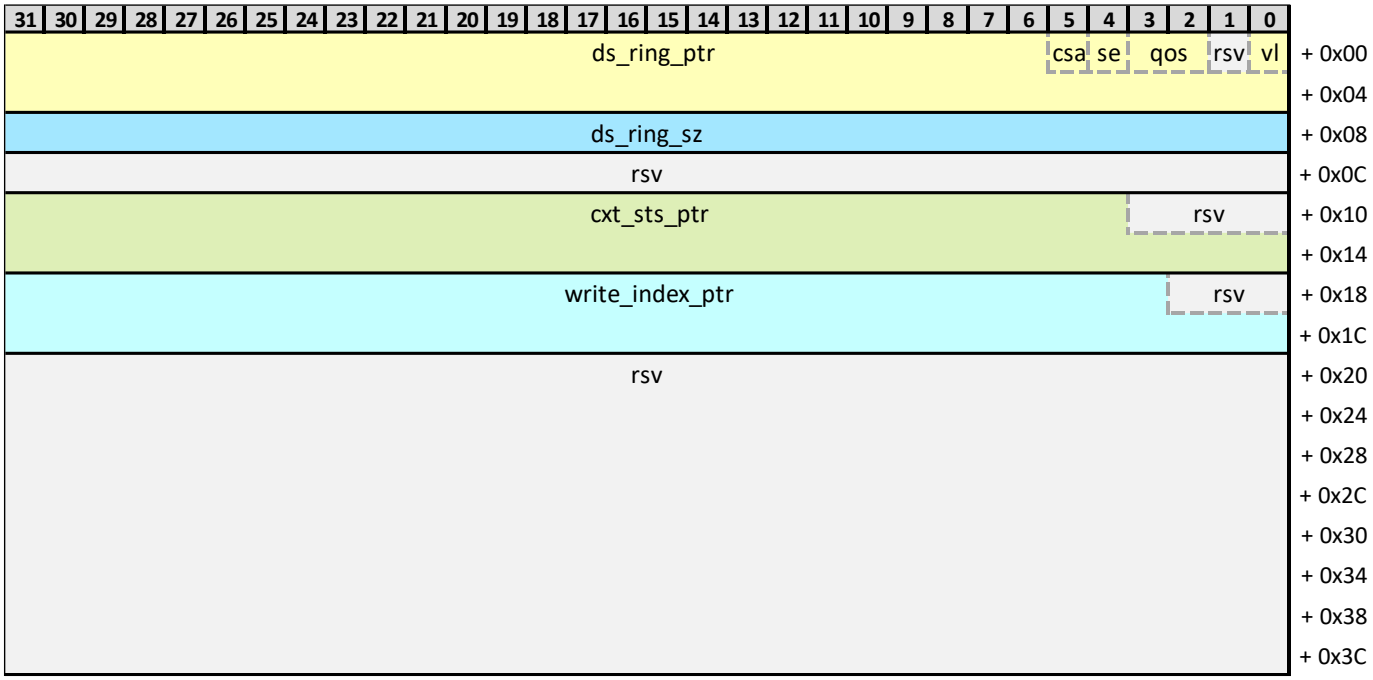


Table 3–4: Context Control (CXT_CTL[^3])

<table>
<tr><th>Field</th><th>Bits</th><th>Subfield</th><th>Description</th></tr>
<tr><td rowspan="6">u64 ds_ring_ptr;</td><td>000</td><td>vl</td><td>Valid. When 1, indicates the other bits in this data structure are valid. When 0, all other bits in this data structure shall be ignored.</td></tr>
<tr><td>001</td><td>rsvd</td><td>Shall be set to zero.</td></tr>
<tr><td>003:002</td><td>qos</td><td>Quality of service indicator. The usage of this field is implementation specific.
<table>
<tr><th>Value</th><th>Definition</th></tr>
<tr><td>00b</td><td>Urgent</td></tr>
<tr><td>01b</td><td>High</td></tr>
<tr><td>10b</td><td>Medium</td></tr>
<tr><td>11b</td><td>Low</td></tr>
</table></td></tr>
<tr><td>004</td><td>se</td><td>Sequential Consistency Hint. 1=Descriptor ring is expected to set the S bit in most descriptors.</td></tr>
<tr><td>005</td><td>csa</td><td>Completion-Status Mode Availability. See "4.4.2, Completion-Status Modes" for details. This field is informational only; the SDXI function takes no action on it.
0 = Both atomic completion-status mode and simple completion-status mode are available to the context.
1 = The context shall only use simple completion-status mode.</td></tr>
<tr><td>063:006</td><td>ds_ring_ptr</td><td>Pointer to the start of the descriptor ring. This points to a 64-byte aligned region of memory</td></tr>
<tr><td>u32 ds_ring_sz;</td><td>095:064</td><td></td><td>Indicates the maximum unsigned number of descriptors that can be placed in the descriptor ring. The size of the descriptor ring in bytes is calculated as (ds_ring_sz * 64). A value of zero is illegal – a ring must have at least one descriptor. A value of 0xFFFF_FFFF indicates a maximum of (2**32)-1 descriptors.
Software shall ensure that:
- Let max_ds = 2**(MMIO_CAP0.max_ds_ring_sz + 10)
- Then CXT_CTL.ds_ring_sz <= min((2**32)-1, max_ds)</td></tr>
<tr><td>u8 rsvd_0[4];</td><td>127:096</td><td></td><td>Shall be set to zero.</td></tr>
<tr><td rowspan="2">u64 cxt_sts_ptr;</td><td>131:128</td><td>rsvd</td><td>Shall be set to zero.</td></tr>
<tr><td>191:132</td><td>cxt_sts_ptr</td><td>Pointer to the Context Status (CXT_STS) data structure which includes the read index value. This points to a 16B aligned region of memory.</td></tr>
<tr><td rowspan="2">u64 write_index_ptr;</td><td>194:192</td><td>rsvd</td><td>Shall be set to zero.</td></tr>
<tr><td>255:195</td><td>write_index_ptr</td><td>Pointer to descriptor ring write index. This points to an 8B aligned region of memory.</td></tr>
<tr><td>u8 rsvd_1[32];</td><td>511:256</td><td></td><td>Shall be set to zero.</td></tr>
</table>

3.2.4 Context Status (CXT_STS)

The Context Status contains status information for a single descriptor Context. Privileged software shall expose it to non-privileged software as a read-only structure. When creating the context, software shall initialize the status entry to all-zero prior to making the context valid. Software shall expose the memory containing the Context Status as read-write to the SDXI function using the context address space.

Figure 3-6: Context Status (CXT_STS)

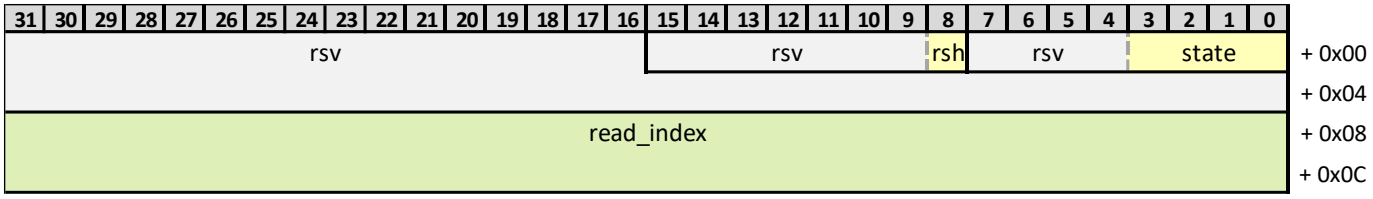


Table 3–5: Context Status (CXT_STS[^3])

Field	Bits	Subfield	Description
u8 state;	003:000	state	Context State. See Table *"Table 3–6: CXT_STS.state Encoding"* for encodings. See section *"4.3.1"* for when this field may be modified.
	007:004	rsvd	This reserved field may be read and written by the SDXI function when it accesses bits [3:0] of the same byte.
u8 misc0;	008	rsh	This hint bit is written only by privileged software to indicate to other software if new descriptors should (rsh = 1) or should not (rsh = 0) be submitted to the context when the ContextState is CXTV_STOPG_SW or CXTV_STOP_SW. See *"4.3.6, Context Ring Submission Hint"* for more details. The SDXI function shall not use nor write this bit. Software shall write this bit with a byte-sized access only.
	015:009	rsvd	Shall be set to zero.
u8 rsvd0_[6];	063:016		Shall be set to zero.
u64 read_index;	127:064		Descriptor ring read index.

Table 3–6: CXT_STS.state Encoding

Value	Definition
0000b	**CXTV_STOP_SW** : Stopped by software using a local-function DSC_CXT_STOP operation on the context.
0001b	**CXTV_RUN** : Running.
0010b	**CXTV_STOPG_SW** : Stopping in response to software using a local-function DSC_CXT_STOP operation on the context.
0100b	**CXTV_STOP_FN** : Stopped either by a transition of the function state from GSV_ACTIVE, or a (PF-to-VF) P2V.DSC_CXT_STOP operation.
0110b	**CXTV_STOPG_FN** : Stopping in response to the function or a P2V.DSC_CXT_STOP operation. The function stopping causes are: an error detected in the function; an error in context processing or execution; or a transition of the function state out of GSV_ACTIVE.
1111b	**CXTV_ERR_FN** : Stopped in response to an error detected in the function, an error in context processing, or an error in context execution.
All other values reserved.	

3.2.5 Access Key (AKey) Table Entry (AKEY_ENT)

The AKey table is a 4-Kbyte aligned contiguous memory structure composed of AKey table entries. The table may be any power-of-2 size from 4-Kbytes up to 1-Mbyte. Software controls the size of the table through the context level 1 entry structure.

A descriptor may reference any AKey associated with the same ring context. The AKey table entries encode all of the valid address spaces, PASIDs and interrupts available to the context.

The AKey entry's tgt_sfunc field specifies the target function within the SDXI function group which owns data buffers and interrupts associated with the AKey table entry. The tgt_sfunc field is set to the target function's value of MMIO_CAP0.sfunc. For PCIe implementations, the tgt_sfunc field is an opaque identifier for the target SDXI function which is used when accessing the data buffer and issuing MSI or MSI-X interrupts.

The tgt_sfunc encoding of 0 is used to indicate the target resource belongs to the same function executing the descriptor. Only the 0 encoding may be used to access local resources.

A non-zero tgt_sfunc value indicates that a remote function owns the target resource. See *"3.3.1, SDXI Function Group"* for more details.

Figure 3-7: AKEY_ENT

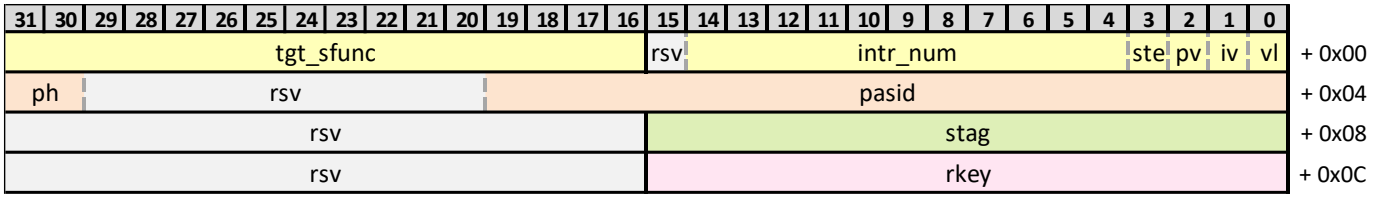


Table 3–7: AKey Table Entry (AKEY_ENT[^3])

Field	Bits	Subfield	Description
u16 intr_num;	000	vl	Valid. When 1, indicates the other bits in this data structure are valid. When 0, all other bits in this data structure shall be ignored.
	001	iv	Interrupt Valid. When 1 and tgt_sfunc = 0, the intr_num field is valid. When tgt_sfunc ≠ 0 this bit is reserved and must be set to 0
	002	pv	PASID Valid. When 1 and tgt_sfunc = 0, the PASID field contains valid information. When tgt_sfunc ≠ 0 this bit is reserved and must be set to 0
	003	ste	Steering Enable. When 1 and tgt_sfunc = 0, memory requests referencing this AKey table entry are enabled to include Data Steering Hint (DSH) information when requested through the descriptor attr.coh_ctl field. When 0 and tgt_sfunc = 0, DSH is disabled for memory requests referencing this AKey table entry even when requested through the descriptor attr.coh_ctl field. When tgt_sfunc ≠ 0 this bit is reserved and must be set to 0. See *"3.6, Data Steering Hints (DSH)"* for more details.
	014:004	intr_num	Interrupts generated using this AKey table entry are issued using the MSI or MSI-X entry corresponding to intr_num. When tgt_sfunc ≠ 0 this field is reserved, and the Target Function's RKey entry specifies which MSI or MSI-X entry to use
	015	rsvd	Shall be set to zero.
u16 tgt_sfunc;	031:016		tgt_sfunc, see above discussion.
u32 pasid;	051:032	pasid	PASID value used for requests using this AKey table entry.
	061:052	rsvd	Shall be set to zero.
	063:062	ph	Processing Hint. When ste is "1", tgt_sfunc = 0, this field supplies information used as part of DSH. When tgt_sfunc ≠ 0 this field is reserved and must be set to 0. The Target Function's RKey table entry is used to supply DSH information. See *"3.6, Data Steering Hints (DSH)"* for more details.
u16 stag;	079:064		Steering Tag. When ste is "1", tgt_sfunc = 0, this field supplies information used as part of DSH. When tgt_sfunc ≠ 0 this field is reserved and must be set to 0. The Target Function's RKey table entry is used to supply DSH information. See *"3.6, Data Steering Hints (DSH)"* for more details.
u8 rsvd_0[2];	095:080		Shall be set to zero.
u16 rkey;	111:096		Specifies the RKey value used to access another function's data buffer or interrupt. This field is only valid if tgt_sfunc is non-zero. See above discussion for more details.
u8 rsvd_1[2];	127:112		Shall be set to zero.

3.3 SDXI Cross-Function Access

There can be multiple interacting SDXI functions in a system if supported by the SDXI implementation. Typically, a PCIe function can only access address spaces mapped using its own Requester ID. A unique capability of an SDXI function is the optional ability to access address spaces mapped by the Requestor ID of another SDXI function.

When executing a descriptor operation, the SDXI function determines the address space for each specified data buffer or interrupt target by using the associated AKey value to reference an AKey table entry. This entry provides the tgt_sfunc which identifies another SDXI function within the same SDXI Function Group. When the tgt_sfunc field is non-zero, the target resource belongs to a remote function whose MMIO_CAP0.sfunc register value matches tgt_sfunc.

The Receiver Access Key (RKey) mechanism described in *"3.3.2, Receiver Access Key (RKey) Table"* provides access control over cross-function requests.

3.3.1 SDXI Function Group

When an SDXI Function returns MMIO_CAP1.rkey_cap as "1", the function supports SDXI cross-function access and the RKey protection mechanism; otherwise, these mechanisms are not supported. Note that SDXI cross-function access when supported is only possible between SDXI functions that belong to the same SDXI Function Group. An SDXI PF and its associated VFs always belong to the same SDXI function group. An SDXI function group may contain multiple PFs and VFs. A system may contain multiple disjoint SDXI function groups. Software may enumerate SDXI Function groups using the MMIO_GRP_ENUM register in each SDXI function; this mechanism only supports a single thread of enumeration. An example of the enumeration flow is given in "*Figure 3-8: Function Group Enumeration Example*". Further methods of coordination and configuration between SDXI functions within an SDXI function group are outside the scope of this specification.

In a cross-function implementation of SDXI, all functions connected within an SDXI function group reflect the same MMIO_GRP_ENUM.probe value written into any group member's copy of this register. Provided that software first initializes all probe fields to "0", software may identify all functions within a function group by writing "1" into MMIO_GRP_ENUM.probe field in one PF, determining that the write has propagated, and then scanning for the same value in other SDXI functions. Software may assign an ID to each function group and record it in the MMIO_CTL0.fn_grp_id field of each associated PF separately for later use; the fn_grp_id field does not propagate among functions. In a VF, the field is read-only and reflects the value of the associated PF.

When the MMIO_GRP_ENUM register is written in one PF, the value of its "probe" field shall subsequently be propagated and reflected in the MMIO_GRP_ENUM.probe field of all other PFs within the same function group within an implementation-specific period of time. Until the "probe" field fully propagates, reading of the the probe field in any other function within the group may return the old or new value of the field.

The SDXI function provides the MMIO_GRP_ENUM.busy field to determine write propagation. Software shall write the "busy" field to a 1 when writing to the MMIO_GRP_ENUM register. An SDXI function may ignore propagating the probe field if the busy field is written as "0"; software shall avoid and not rely upon such behavior. The SDXI function shall clear the "busy" field of that PF when the write of the "probe" field has fully propagated to all other functions within the function group. Software determines the write has propagated by looping on the value of that PF's "busy" field until it is read as "0". The "busy" field itself does not propagate.

When software writes MMIO_GRP_ENUM in one PF, it should not write the same register in any other SDXI function until the first write fully propagates.

The MMIO_GRP_ENUM register is only used to identify functions within the same SDXI function group. It is not referenced by other SDXI registers or data structures. In a VF, the field is read-only and reflects the value of the associated PF.

Figure 3-8: Function Group Enumeration Example

```
/* SDXI Fn Group Enumeration Example Pseudo-code
    Let PF_LIST be an array of fn_struct, one for each PF. Let VF_LIST be an array of fn_struct, one for each VF.
    This can be determined for each SDXI using MMIO_CAP0.vf. In practice, a real OS would use something
    different. Let each fn_struct have the fn's requestor_id, mmio_base_ptr,  pf_flag (if pf then 1 else 0), and the SW
    assigned grp_id.
    Let read & write of a fn's mmio space be done by the functions: get_fn_mmio(fn_struct, offset) -- returns the
    location value; and set_fn_mmio(fn_struct, offset, value) -- writes the location.
    Let ID_LIST contain a list of all known GRP_ID values. Let clear_ids(id_list) clear the list and add_id(id_list, id)
    add to the list.  Define get_new_id(id, id_list) to return a new unique 32-bit grp_id.
*/


set_grp_enum(fn, probe){
   // Good Hygiene: Wait for probe-propagating to be false
   while (get_fn_mmio(fn, MMIO_GRP_ENUM) & 1);
   set_fn_mmio(fn, MMIO_GRP_ENUM, (probe << 1) | 1);
   // Wait for probe-propagating to be false
   while (get_fn_mmio(fn, MMIO_GRP_ENUM) & 1);
}


set_grp_id(fn, id){
   id = (get_fn_mmio(fn, MMIO_CTL0) & 0xFFFF_FFFF) | (id << 32);
   set_fn_mmio(fn, MMIO_CTL0, id);
}


get_grp_id(fn){
   return get_fn_mmio(fn, MMIO_CTL0) >> 32;
}

// S0: Initialize PFs and tracking info
clear_ids(ID_LIST);


for (i = 0; i < NUMBER_OF_PF; i++){
   pf = PF_LIST[i];
   pf.grp_id = 0;
   set_grp_id(pf, 0);
   set_grp_enum(pf, 0);
}


// S1: Init VF data and add Grp_IDs from VFs not of PFs in PF_LIST
for (i = 0; i < NUMBER_OF_VF; i++){
   vf = VF_LIST[i];
   vf.grp_id = get_grp_id(vf);
   if (vf.grp_id != 0){ add_id(ID_LIST, vf.grp_id); }
}
```

Figure 3-9: Function Group Enumeration Example (cont)

```
// S2: Probe through all PFs and assign GRP IDs
for (i = 0; i < NUMBER_OF_PF; i++){
   pf0 = PF_LIST[i];
   if (pf0.grp_id != 0){ continue; }  // pf already enumerated

   // S3: pf is not yet enumerated, assign new grp_id & probe
   pf0.grp_id = get_new_id(ID_LIST);
   set_grp_id(pf0, pf0.grp_id);
   set_grp_enum(pf0, 1);

   // S4: Probe remaining PFs for FN Grp membership
   for (j = i+1; j < NUMBER_OF_PF; j++){
      pf1 = PF_LIST[j];
      if (pf1.grp_id != 0){ continue; }  // Already enumerated

      // S5: If probe detected, must be part of the current group
      if ( get_fn_mmio(pf1, MMIO_GRP_ENUM) & 0b10 ){
         pf1.grp_id = pf0.grp_id;
         set_grp_id(pf1, pf0.grp_id);
      }
   }

   // S6: This group is probed, clear the probe bit
   set_grp_enum(pf0, 0);
}


// S7: Scan for VFs of PFs in PF_LIST and record grp_id
for (i = 0; i < NUMBER_OF_VF; i++){
   vf = VF_LIST[i];
   if ( vf.grp_id == 0){ vf.grp_id = get_grp_id(vf); }
}
```

3.3.2 Receiver Access Key (RKey) Table

A (local) SDXI function uses RKey table entries to control remote requesting functions' access to local function resources such as memory and interrupts. Software may use RKey table entries in combination with IOMMU page tables to selectively expose data buffers and interrupts to different requesting functions.

The RKey table is a 4-Kbyte aligned contiguous memory structure composed of RKey table entries. The table may be any power-of-2 size from 4-Kbytes up to 1 Mbyte. Software controls the size of the table through MMIO_RKEY.tbl_sz.

A connection manager specification is proposed for development by the SNIA SDXI TWG post publication of Smart Data Accelerator Interface ("SDXI") Specification v1.0. Please refer to it for mechanisms to allocate, configure, and exchange RKeys between functions.

3.3.3 RKey Table Entry

Each RKey table entry is an aligned 16-byte structure with the following format.

Figure 3-10: RKey Table Entry

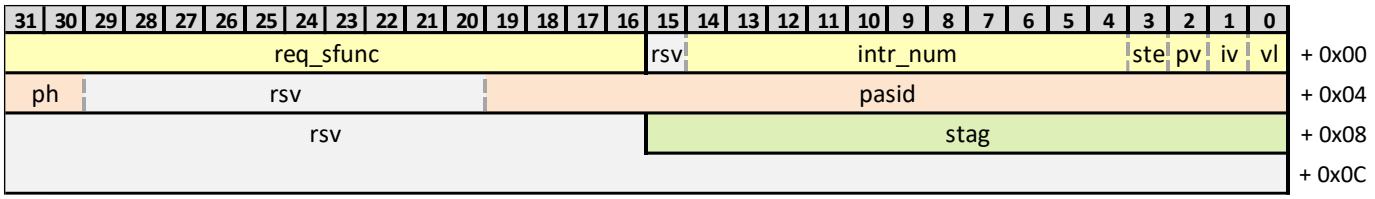


Table 3–8: RKey Table Entry (RKEY_ENT[^3])

Field	Bits	Subfield	Description
u16 intr_num;	000	vl	Valid. When 1, indicates the other bits in this data structure are valid. When 0, all other bits in this data structure shall be ignored and remote requesting functions may not use the RKey value associated with this entry to access data buffers or issue interrupts owned by this function.
	001	iv	Interrupt Valid. When 1, Requesting functions referencing this RKey table entry are permitted to generate interrupt requests via this function. When 0, interrupt requests generated by referencing this RKey table entry are aborted.
	002	pv	PASID Valid. When 1, the PASID field contains valid information.
	003	ste	Steering Enable. When 1, memory requests referencing this RKey table entry are enabled to include DSH information when requested through the descriptor attr.coh_ctl field. When 0, DSH is disabled for memory requests referencing this RKey table entry even when requested through the descriptor attr.coh_ctl field. See *"3.6, Data Steering Hints (DSH)"* for more details.
	014:004	intr_num	Interrupts generated using this RKey table entry are issued using the target function's MSI or MSI-X entry corresponding to intr_num. When valid, intr_num may be used as part of PCIe TPH. See *"8.3, Mapping SDXI DSH to PCIe TLP Processing Hints (PCIe TPH)"* for more details.
	015	rsvd	Shall be set to 0.
u16 req_sfunc;	031:016		req_sfunc specifies the "sfunc" value of the remote requesting function expected to reference this RKey table entry.
u32 pasid;	051:032	pasid	PASID value used to access data buffers using this RKey entry
	061:052	rsvd	Shall be set to zero.
	063:062	ph	Processing Hint. When "ste" is "1", this field supplies information used as part of DSH. When "ste" is "0", this field is reserved. See *"3.6, Data Steering Hints (DSH)"* for more details.
u16 stag;	079:064		Steering Tag. When ste is "1", this field supplies information used as part of DSH. When "ste" is "0", this field is reserved. See *"3.6, Data Steering Hints (DSH)"* for more details.
u8 rsvd_0[6];	127:080		Shall be set to zero.

3.3.4 Receiver Access Key (RKey) Processing

When an operation's AKey table entry specifies a "tgt_sfunc" value of zero (i.e. the access is within a function), RKey processing for that access or interrupt is skipped. When "tgt_sfunc" is non-zero (i.e. the access is Cross-Function), RKey processing is attempted using the following checks below.

1. If the Target function does not belong to the same function group as the requesting function, abort the remote access or interrupt.
2. Requesting function passes its own MMIO_CAP0.sfunc value along with the AKey table entry's RKey value to the target function indicated by "tgt_sfunc".
3. If the target function has MMIO_CAP1.rkey_cap as "0", abort the remote access or interrupt.
4. Target function uses the supplied RKey value as an index into its RKey table to obtain the associated RKey table entry. If the RKey table is not enabled (MMIO_RKEY.tbl_en is "0"), or the RKey table entry is not valid, abort the remote access or interrupt.
5. Target function checks if the RKey table entry req_sfunc matches the requester "sfunc". If not, abort the remote access or interrupt.
6. For an interrupt operation, the target function checks if the RKey table entry has "iv" as "1". If not, abort the remote interrupt. When "iv" is "1", the intr_num field in the RKey table entry specifies which MSI or MSI-X index in the target function is used.
7. For a memory access operation, if the RKey table entry has "pv" as "0", the associated memory accesses are issued by the target function with no PASID applied. When "pv" is "1", the associated memory accesses are issued by the target function with a PASID applied using the "pasid" field from the RKey table entry.
8. For a memory access operation, if the RKey table entry has "se" as "0", the associated memory accesses are issued with no SteeringTag or PH applied. When "se" is "1", a SteeringTag and PH are applied to the associated memory accesses using the "stag" and "ph" fields in the RKey table entry.

The requesting SDXI function will log failed remote accesses as Data Buffer errors (ERRV_DSC_BUF) regardless of the reason for the failure. The requesting SDXI function may use any appropriate value for the error sub-step, including 0 (‘Other’). Data Buffer failures will stop the requesting context. Examples of such failures include communication errors between the requester and target, the target being off-line, access rights failures at the target, address translation errors at the target, or data access errors at the target.

The target SDXI function will not log problems with received requests such as: illegal or disabled RKey indexes; mismatched sfunc values; or attempted accesses while the function or RKey processing is disabled. The target SDXI function also will not log issues encountered while accessing data buffers on behalf of a remote requester such as address translation, poison consumption, or time out errors. Instead, these issues are logged in the requesting function.

If a target SDXI function encounters the following type of error trying to access its own RKey table then the SDXI function will log a “ERRV_FN_RKEY” error and continue execution: an address translation error; poisoned RKey table entry; and illegal data within an Rkey entry with the “vl” bit set to 1. If an SDXI remote request is rejected due to the RKey lookup failure, the requestor will also log the failure as a ERRV_DSC_BUF, and will halt the impacted context.

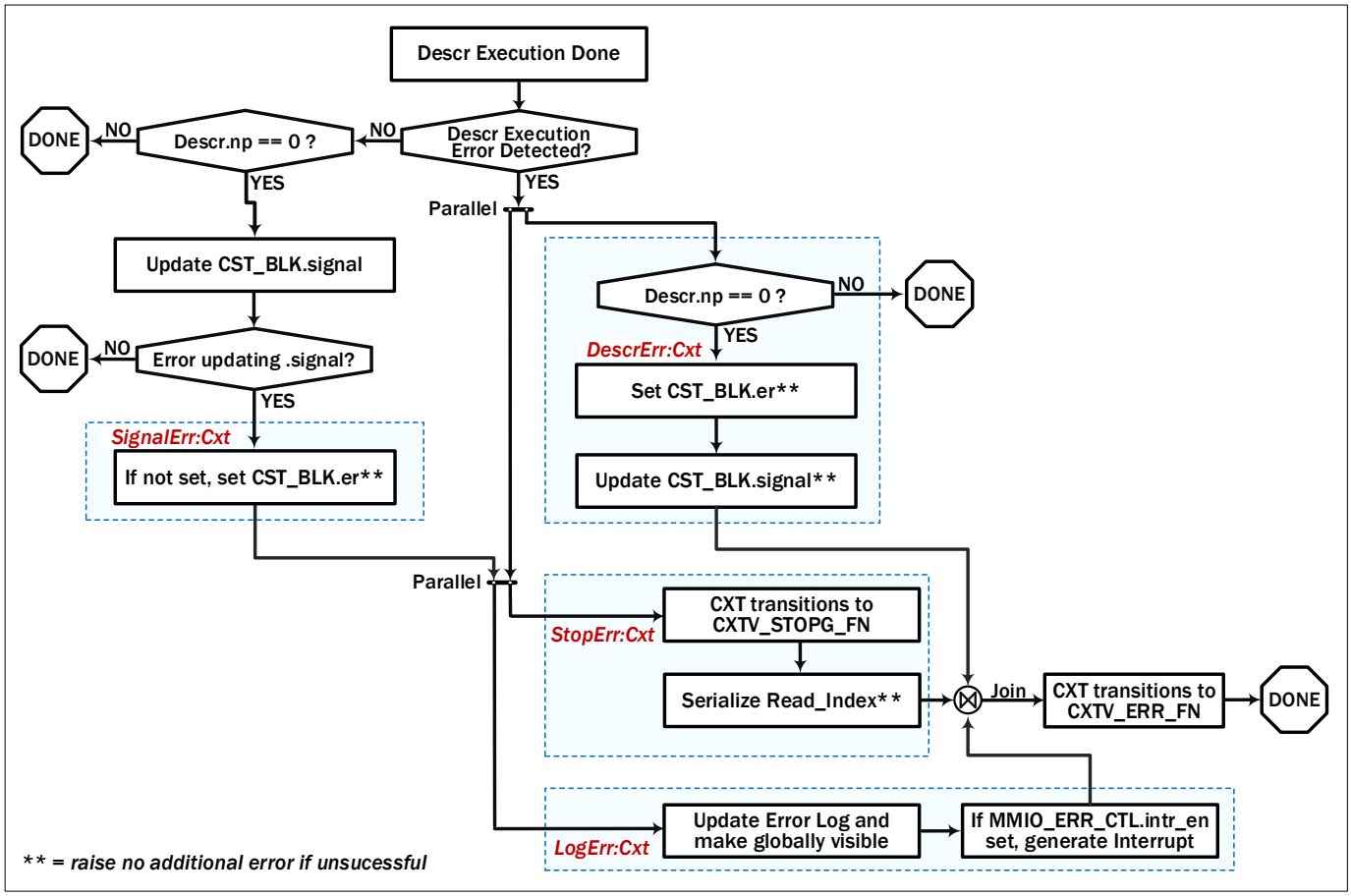
3.4 Error Log

Whenever the SDXI function detects a function-wide or context-specific error, it performs specific actions to contain the error, reports it to software through an entry in the function's Error Log message ring, and signals an error in an associated descriptor's completion status block if relevant. A summary of the type of errors, their logging, and the associated error containment actions follows.

1. ***Invalid Context (Invalid:Cxt):*** the function has evaluated a context that is in CXTV_INVALID state. The function shall skip further operation on the context. Unless explicitly required by an action or operation to suppress or signal an error, an SDXI implementation should suppress signaling errors for Invalid:Cxt. See *"4.2.1, S1: CXTV_INVALID State"* for more details.
2. ***Logging A Context Error (LogErr:Cxt):*** the function generates an error log entry when it detects a context error which can include DescrErr:Cxt and SignalErr:Cxt errors. After the error log entry is globally visible, the function shall signal an interrupt if MMIO_ERR_CTL.intr_en is set. This action can start in any order or in parallel with DescrStatus:Cxt, SignalErr:Cxt, and StopErr:Cxt but shall complete before StopErr:Cxt does. See *"Figure 3-11: Context Descriptor Or Operation Error Flow"* for an overall flow of the error handling.
3. ***Updating CST_BLK.er (DescrErr:Cxt):*** If the CST_BLK is enabled for a descriptor ("np" is "0") and a descriptor error is detected in processing or executing a descriptor, then CST_BLK.er is set and made globally visible before the function updates CST_BLK.signal. (Note, errors encountered in updating CST_BLK for this action do not start any additional actions.) This action can start in any order or in parallel with LogErr:Cxt, SignalErr:Cxt, and StopErr:Cxt but shall complete before StopErr:Cxt does. See *"Figure 3-11: Context Descriptor Or Operation Error Flow"* for an overall flow of the error handling.
4. ***Error In Updating CST_BLK.signal (SignalErr:Cxt):*** If an error occurs when updating CST_BLK.signal, then the SDXI function shall set CST_BLK.er, if not already set, and make it globally visible. (Note, an error in setting CST_BLK.er for this action does not start any additional actions.) This action will also start in any order or in parallel LogErr:Cxt and StopErr:Cxt actions, if not already started. This action shall complete before StopErr:Cxt does. See *"Figure 3-11: Context Descriptor Or Operation Error Flow"* for an overall flow of the error handling.
5. ***Stopping A Context Due to Error (StopErr:Cxt):*** the function initiates a background context stop action for a context in the CXTV_RUN state in response to an Invalid:Cxt (when indicated), DescrErr:Cxt, or SignalErr:Cxt. The stop action terminates processing of new descriptors and waits for existing ones to complete. Once the context is stopped, the function serializes Read_Index and then transitions the context to CXTV_ERR_FN state. This action starts in any order or in parallel with LogErr:Cxt, DescrErr:Cxt, and SignalErr:Cxt but shall be completed only after them. This completion order ensures that when software sees the context in CXTV_ERR_FN state, all relevant error log records and CST_BLK entries are updated and globally visible. See *"Figure 3-11: Context Descriptor Or Operation Error Flow"* for an overall flow of the error handling.
6. ***Function-Wide Error (HaltErr:Fn):*** the function initiates an error halt action in response to an uncorrectable error that prevents further safe operation of the function across all of its associated contexts. Examples include the detection of invalid configuration of MMIO registers (due to privileged software) as well as internal logic errors inside the function that lead to a HItErr:Fn action. For this action the function does the following ordered steps.
    a. In any order or in parallel: the function halts itself; and makes best effort to log a function error (LogErr:Fn) and make it globally visible. It is implementation dependent if the function performs any context-specific state changes or stop actions when halting the function; software shall not rely upon it.
    b. When the above completes, the function changes MMIO_STS0.fn_gsv to GSV_ERROR. This order ensures that when software reads the error log immediately after detecting the transition to GSV_ERROR, it will receive all available information.

7. ***Logging A Function Error (LogErr:Fn):*** the function attempts to generate an error log entry when an error halt action (HaltErr:Fn) is initiated.

Figure 3-11: Context Descriptor Or Operation Error Flow



Note that as a consequence of the above error flow for a context error, the SDXI function shall ensure that the error's effect upon the error log, the descriptor's CST_BLK, Read_Index serialization, and the context's state are synchronized when the context state transitions to CXTV_ERR_FN -- not before. Therefore, it is important that regardless of the manner software first becomes aware of a context error, it should wait for the context to enter the CXTV_FN_ERROR state before examining error information.

Each SDXI function supports a single, function-wide error log message ring in memory for all the contexts of that function; the ring is configured through MMIO space. The error log and its indexes are structured similar to an SDXI descriptor ring and operate in the manner below.

1. MMIO_ERR_CFG points to the base of the error log. Each error log index corresponds to an aligned 64-byte entry in the error log ring.
   a. For a given error log index, its address in memory is given by
      i. log_sz = 2**(MMIO_ERR_CFG.sz +12);
      ii. log_bs = MMIO_ERR_CFG & ~0xFFF;
      iii. address = log_bs + ((index * 64) % log_sz);
   b. The indices are not expected to reach or exceed (2**64) - 1 in practice.

2. MMIO_ERR_WRT indicates the next available error log index that can be written by the function.
3. MMIO_ERR_RD indicates the index of the first error log entry not yet consumed by software. As software consumes error log entries, it advances the index in MMIO_ERR_RD. For each error log entry consumed, software takes appropriate remediation action. As the ring is finite in size, software shall consume error log entries in a timely manner lest the error log overflows.
4. As the function writes more error log entries, it advances the index in MMIO_ERR_WRT. The error log is empty when MMIO_ERR_WRT is equal to MMIO_ERR_RD.
5. For each error detected when the error log is enabled, the SDXI function writes a 64-byte aligned error log entry.
    a. If there is insufficient space in the error log for the function to write an error log entry, the error log shall overflow.
    b. When an overflow event occurs, the MMIO_ERR_STS.ovf bit shall be set, MMIO_ERR_WRT shall not advance, and error logging shall be stopped. Contexts may continue to run even when the error log is full or in the overflow state, however detailed error information for new errors will be lost.
    c. Note that the error log can overflow even if there are available entries in the log when the number of entries in an error log sequence exceed the available entries.
6. After writing an error log entry successfully, MMIO_ERR_WRT will be advanced by 1, MMIO_ERR_STS.sts will be set to “1”, and an error log interrupt shall be generated if MMIO_ERR_CTL.intr_en is set to "1". If MSI or MSI-X are enabled, vector 0 will be used for the error log interrupt.

When the function logs a context descriptor error (LogErr:Cxt), it sets ERRLOG_HD_ENT.cv to "1". During descriptor processing and before execution, errors may be detected and logged at different processing steps for a single descriptor resulting in multiple error logs. Only one error may be logged for errors occurring during the execution phase of the descriptor operation. Refer to *"5.3, Descriptor Processing"* for more details.

Even though descriptors may reference data buffers in different address spaces and may detect errors when accessing those buffers, the errors are logged with the function hosting the context.

Note that the function may simultaneously consume and execute multiple descriptors from multiple contexts. The order in which errors are detected and their error log entries are written is implementation-dependent. However, each error log entry sequence shall be written by the function in its entirety indivisibly with respect to other error log entries.

Errors may be reported on DMA write operations. While DMA writes are posted in PCI-Express and do not return any status, write response status may be available in implementations that do not connect using a physical PCIe link.

3.4.1 Error Log Header Entry (ERRLOG_HD_ENT)

The SDXI function reports the details of a specific error event by writing an error log header entry (ERRLOG_HD_ENT) to the error log. Error log header entries are formatted as described in *"Table 3-9: Error Log Header Entry (ERRLOG_HD_ENT[^3])"* and shown in "*Figure 3-12: Error Log Header Entry (ERRLOG_ENT)"*. (The entry format is similar to an SDXI descriptor but is made distinct by a "type" field value of "0x7F7".) The SDXI function constructs the error log entry in the following way.

1. The SDXI function shall set the "vl" field to "1" and the "type" field to "0x7F7".
2. The SDXI function shall set the "re" field, which indicates whether the error was serious enough to stop a context or the entire SDXI function.
3. The function shall set the "step" field to allow system software to quickly distinguish between errors resulting from the producer and errors likely resulting from system software, the SDXI function, memory, or other hardware components. The "step" field encodings are shown in *"Table 3-10: (Flagged) Processing Step"*.
4. The function should set the "sub_step" field to the appropriate value whenever this information is known; however, the function may report "0" ("other") otherwise.
5. When relevant (see table 3-10), the function should identify the failing context number by setting the "cxt_num" field and setting the "cv" field to "1".
6. When relevant (see table 3-10), the function should identify the failing descriptor index by setting the "dsc_index" field and setting the "div" field to "1".
7. When relevant (see table 3-10), the function should identify one of the failing buffers or AKEYs by setting the "buf" field and setting the "bv" field to "1".
8. To assist in offline debug analysis, the function may provide an error class ("err_class") and vendor specific information ("vendor") applicable to the detected error. The code value returned is implementation dependent and not intended for use by system software for online remediation. The error class codes form a hierarchy as described in *"Table 3 11: Error Class Hierarchy"*. Although, an SDXI function implementation may choose to always return a code of 'Generic Error' (0), it is recommended that the function make best effort to return something more meaningful.

Figure 3-12: Error Log Header Entry (ERRLOG_ENT)

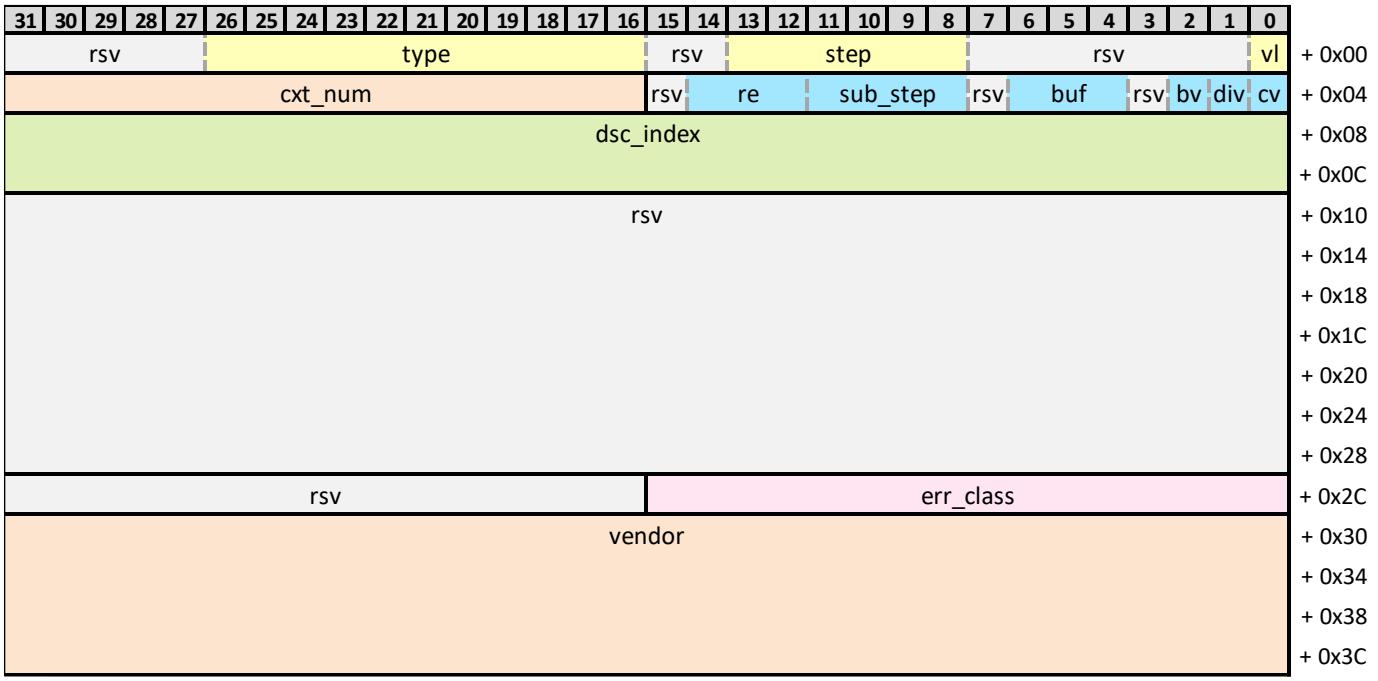


Table 3-9: Error Log Header Entry (ERRLOG_HD_ENT[^3])

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. When 1, indicates the other bits in this data structure are valid. When 0, all other bits in this data structure shall be ignored.
	007:001	rsvd	Shall be set to "0"
	013:008	step	Identifies the SDXI function processing step that encountered an error. See “Table 3-10: (Flagged) Processing Step”
	015:014	rsvd	Shall be set to 0.
	026:016	type=0x7F7	Descriptor type 0x7F7 = ERRLOG_ENT. See section "6.1.1".
	031:027	rsvd	Shall be set to 0.
u16 misc0;	32	cv	Indicates if the cxt_num field is valid. See See “Table 3-10: (Flagged) Processing Step” for when “cv” must be set.
	33	div	Indicates if the dsc_index field is valid. See See “Table 3-10: (Flagged) Processing Step” for when “div” must be set.
	34	bv	Buffer Valid: indicates if the “buf” field is valid. See See “Table 3-10: (Flagged) Processing Step” for when “bv” must be set.
	35	rsvd	Shall be set to 0.
	38:36	buf	For SDXI descriptors with multiple Buffers or Akey indexes, identifies which instance was involved in the error. If an SDXI function implementation detects more than one akey/buffer failure while processing a descriptor, the implementation shall choose one of them to log. 0 = Buffer0 or Akey0 1 = Buffer1 or Akey1 2-7 = reserved for future use
	39	rsvd	Shall be set to 0.
	43:40	sub_step	Identifies the specific sub-step that failed 0 = Other (or Internal Error) (May be used when sub-step is unknown) 1 = Address Translation Failure 2 = Data Access Failure (i.e. Read/Write/Atomic Access) 3 = Data Validation Failure (i.e. data content is invalid) -15 = reserved for future use
	46:44	re	SDXI Function Reaction to the Error 0 = Informative Entry, nothing stopped 1 = SDXI Context Stopped (“cv” and “cxt_num” shall be valid). 2 = SDXI Function Stopped 3-7 = Reserved for future use
	47	rsvd	Shall be set to 0.
u16 cxt_num;	063:048		Context number associated with the error log entry when cv = “1”.
u64 dsc_index;	127:064		Descriptor index of descriptor that failed when "div" ="1".
u8 rsvd_0[28];	351:128		Shall be set to 0.
u16 err_class;	367:352		Provides an error classification code for offline debug analysis. See discussion above.
u8 rsvd_1[2];	383:368		Shall be set to 0.
u32 vendor[4];	511:384		These fields are available for SDXI functions to record additional vendor defined debugging information relating to the error, such as telemetry relating to internal errors.

Table 3-10: (Flagged) Processing Step

Encoding (Enc) & Types		Req'd Fields			SDXI Function Action (Response)
Enc	Processing Step	cv	div	bv	
0	reserved	n/a	n/a	n/a	n/a
1	Internal Error (ERRV_INT)	X	0	0	Function Stop or Context Stop ('cv' shall be 1 if Context Stop).
2	Context Level 2 Table Entry – Translate, Read, Validate. (ERRV_CXT_L2)	1	0	0	Function Stop or Context Stop.
3	Context Level 1 Table Entry – Translate, Read, Validate. (ERRV_CXT_L1)	1	0	0	Function Stop or Context Stop.
4	Context Control – Translate, Read, Validate. (ERRV_CXT_CTL)	1	0	0	Function Stop or Context Stop.
5	Context Status – Translate, Access, Validate. (ERRV_CXT_STS)	1	0	0	Function Stop or Context Stop.
6	Write_Index – Translate, Read, Validate. (ERRV_WRT_IDX)	1	0	0	Context Stop.
7	Descriptor Entry – Translate, Access, Validate. (ERRV_DSC_GEN)	1	1	0	Context Stop.
8	Descriptor CST_BLK – Translate, Access, Validate. (ERRV_DSC_CSB)	1	1	0	Context Stop.
9	Atomic Return Data – Translate, Access. (ERRV_ATOMIC)	1	1	0	Context Stop.
10	Descriptor: Data Buffer – Translate, Access. (ERRV_DSC_BUF)	1	1	1	Context Stop.
11	Descriptor AKey Lookup – Translate, Access, Validate. (ERRV_DSC_AKEY)	1	1	1	Context Stop.
12	Function RKey Lookup – Translate, Read, Validate. (ERRV_FN_RKEY)	0	0	0	Informative Entry, nothing stopped
13-63	reserved	n/a	n/a	n/a	n/a

Table 3-11: Error Class Hierarchy

Bit layout and definitions for err_class				
[15:12]	[11:08]	[07:04]	[03:00]	
0x0000	Generic SDXI Error			
	0x1000	SDXI Internal Function Error		
	0x2000	Generic Logical Error		
		0x2100	Unsupported Field Encoding	
		0x2200	Non-zero reserved field used	
		0x2300	Specification, Implementation, or instance limit exceeded	
			0x2310	max_buffer limit exceeded
			0x2320	illegal or invalid AKey Index
			0x2330	illegal or invalid context index
			0x2340	illegal or invalid descriptor ring size
			0x2350	illegal values for Read_Index or Write_Index (overflow or underflow)
			0x2360	illegal PASID (implementation limit or format)
			0x2370	illegal intr_num (implementation limit or format)
		0x2400	Unsupported descriptor type/subtype encoding	
	0x3000	Generic Memory Access Error		
		0x3100	Memory Access Permission Error	
			0x3110	PCIe Unsupported Request Error
			0x3120	PCIe Completer Abort Error
		0x3200	Memory Access Data Poison Error	
		0x3300	Memory Access Time-out	
		0x3400	Memory Translation Error	
			0x3410	PCIe Translation Completion Unsupported Request Error
			0x3420	PCIe Translation Completion Completer Abort Error
		0x3500	Page Request Error or Time-out	
			0x3510	PCIe PRG Invalid Request Response Error
			0x3520	PCIe PRG Failure Response Error
All other encoding reserved				

3.4.2 Error Log Initialization

To initialize or re-initialize the error log, the following procedure shall be used by software.

1. Clear MMIO_ERR_CFG.en to "0".
2. Clear MMIO_ERR_STS to "0".
3. To hold the error log, allocate a 4-KB aligned contiguous memory buffer whose size is a power-of-two of at least 4-KB. Initialize the buffer to zero.
4. Set MMIO_ERR_CTL.intr_en to "1" if interrupts on errors are desired.
5. Clear MMIO_ERR_WRT and MMIO_ERR_RD to "0".
6. Program MMIO_ERR_CFG:
    a. Program MMIO_ERR_CFG.ptr to point to the start of the error log memory region.
    b. Program MMIO_ERR_CFG.sz to the appropriate size of the error log.
    c. Set MMIO_ERR_CFG.en to "1" to enable the error log mechanism.

3.4.3 Error Log Processing by Software

Software may use the following set of steps to process the error log.

1. Check MMIO_ERR_STS and perform any required remediation.
2. If MMIO_ERR_STS.sts is "1", then compute read_index; otherwise exit this set of steps.
    - read_index = MMIO_ERR_RD;
3. Clear MMIO_ERR_STS to "0". This clears all error log overflow, error, and status flags. If using an edge triggered interrupt mechanism, the system should be ready to accept another interrupt without loss before this MMIO write to MMIO_ERR_STS.
4. Compute write_index. The MMIO read of MMIO_ERR_WRT must follow step 3 to ensure that either software is working with the latest value of MMIO_ERR_WRT or another interrupt will occur after software completes this entire set of steps.
    - write_index = MMIO_ERR_WRT;
5. If read_index == write_index exit this set of steps.
6. If read_index < write_index, then proceed to the next step; otherwise go to step 11.
7. Compute the following:
    - log_sz = 2**(MMIO_ERR_CFG.sz +12);
    - log_bs = MMIO_ERR_CFG & ~0xFFF;
8. Process the error log entry at this address:
    - address = log_bs + ((read_index * 64) % log_sz);
9. Advance read_index.
10. Loop back to step 6.
11. Write read_index to MMIO_ERR_RD.

3.5 Administrative Context (Context 0)

Administrative operations are used by privileged software to manage the SDXI function and are only supported in Context 0 ("Administrative" Context) within each function. Software shall ensure context 0 is the first context started and the last stopped to ensure proper operation. When context 0 goes into an undefined or error state, software shall stop and restart the function to ensure correct operation.

An SDXI function may only cache private copies of all LVL_L2 data (see *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching"*) associated with context 0 when the context is at the CXTV_RUN state.

An Administrative Context only supports AdminGrp, ConnectGrp and other specified administrative operations; no other operation groups including the DmaBaseGrp operations are supported. A VF Administrative Context may be used to manage contexts associated with that VF. A PF Administrative Context may be used to manage contexts hosted by the PF or contexts hosted by any VF associated with the PF. If a VF encounters a failure while processing a P2V administrative operation, the error will be logged by the VF.

3.6 Data Steering Hints (DSH)

An SDXI Function may allow for the inclusion of DSH on data buffer accesses. One potential use of DSH is to provide a cache injection hint that allows the data buffer to be injected into a CPU cache. The actual encoding, propagation, and use of DSH information is platform specific and outside the scope of this specification.

The mapping of DSH information onto device architecture mechanisms is specific to those architectures. For details on how DSH information maps onto PCIe, please refer to "*8.3, Mapping SDXI DSH to PCIe TLP Processing Hints (PCIe TPH)*".

The context producer may request the inclusion of DSH information on a data buffer access by setting the data buffer's "attr.coh_ctl" field to "10b" in the descriptor.

DSH information is only included when all of the following conditions are true:

- All required bus/device mechanisms to support DSH are present and enabled
- The data buffer's "attr.coh_ctl" field is "10b"
- If the data buffer's "AKEY_ENT.sfunc" field is zero, the "AKEY_ENT.ste" field is set to 1
- If the data buffer's "AKEY_ENT.sfunc" field is non-zero, the data buffer's "RKEY_ENT.ste" field is set to 1

A 2-bit ph field and 16-bit stag field are provided through the data buffer's AKey or RKey table entry.

SDXI implementations are free to apply the requested DSH information on all, some, or none of the accesses to the referenced data buffer.

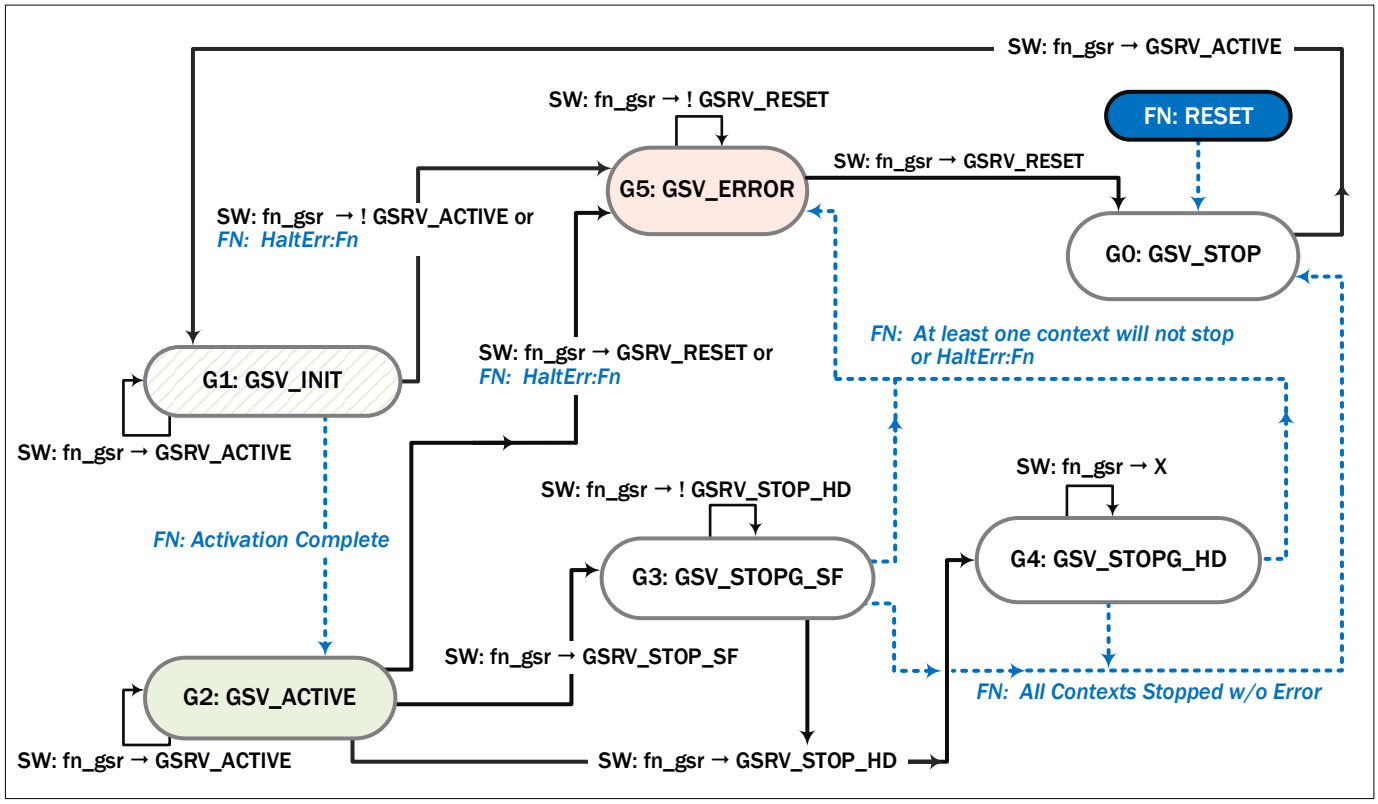
Software is responsible for ensuring that the target of a request containing DSH information supports receiving DSH information. Software is also responsible for ensuring that the bus infrastructure between the requesting SDXI Function and the data buffer target supports the propagation of DSH information.

The reaction of a target to receiving unexpected DSH information or encodings is outside the scope of this specification.

4.1 SDXI Function State

An SDXI function operates serially within a number of basic states indicated by the MMIO_STS0.fn_gsv register and shown in the figure below. Software may use the MMIO_CTL0.fn_gsr register to help transition the function between the various states.

Figure 4-1: SDXI Function States and State Transitions



MMIO_STS0.fn_gsv values	
0b000	GSV_STOP
0b001	GSV_INIT
0b010	GSV_ACTIVE
0b011	GSV_STOPG_SF
0b100	GSV_STOPG_HD
0b101	GSV_ERROR
All Others Reserved	

MMIO_CTL0.fn_gsr	
0b00	GSRV_RESET
0b01	GSRV_STOP_SF
0b10	GSRV_STOP_HD
0b11	GSRV_ACTIVE

4.1.1 G0: GSV_STOP State

The function is idle and not enabled to process any memory data structures or perform DMA operations. All DMA operations from remote functions targeting this function's local address spaces are aborted.

From the GSV_STOP state, software may set MMIO_CTL0.fn_gsr to GSRV_ACTIVE to start the function. The function will attempt to transition through the GSV_INIT state into the GSV_ACTIVE state. Software shall program MMIO_CXT_L2 prior to setting MMIO_CTL0.fn_gsr to GSRV_ACTIVE. Writing any other value to MMIO_CTL0.fn_gsr is ignored.

4.1.2 G1: GSV_INIT State

Upon receiving the GSRV_ACTIVE request while in the GSV_STOP state, the function shall transition into the GSV_INIT state to perform any necessary self-initialization. All DMA operations from remote functions targeting this function's local address spaces are aborted in this state.

If initialization is successful, the function transitions to the GSV_ACTIVE state. As part of the initialization, the SDXI function may validate the contents of the following MMIO registers.

- MMIO_CTL0 and MMIO_CTL2.

The SDXI function initializes the contents of the following MMIO registers.

- MMIO_STS0.

While in this state, if an error occurs during the initialization that prevents the processing of all contexts or software sets MMIO_CTL0.fn_gsr to any value other than GSRV_ACTIVE (which is a ERR_PGM_GEN.field_invalid error), the function shall initiate a HaltErr:Fn action, which includes transitioning MMIO_STS0.fn_gsv to the GSV_ERROR state.

4.1.3 G2: GSV_ACTIVE State

In the GSV_ACTIVE state, the function is enabled to process contexts and descriptors. From this state, privileged software may request the function to stop by writing MMIO_CTL0.fn_gsr to GSRV_STOP_HD to request a hard stop or to GSRV_STOP_SF to request a soft stop.

While in this state, if an error occurs that prevents the processing of all contexts or software sets MMIO_CTL0.fn_gsr to GSRV_RESET (which is a ERR_PGM_GEN.field_invalid error), the function shall initiate a HaltErr:Fn action, which includes transitioning MMIO_STS0.fn_gsv to the GSV_ERROR state.

4.1.4 G3: GSV_STOPG_SF ("Soft Stopping") State

In the GSV_STOPG_SF state, the function shall stop processing new descriptors and wait until all outstanding descriptors complete naturally or until an implementation-defined timeout period has expired at which point any remaining descriptors are aborted. The function shall log errors as necessary for each aborted descriptor. See *"4.3.5, Function and Context Stop Actions"*.

When all contexts have reached their final state, and all previously received MMIO doorbells are fully evaluated (i.e. there are no outstanding data structure reads due to a prior doorbell), the function transitions to the GSV_STOP state.

While in this state, if an error occurs that prevents one or more contexts from stopping, the function shall initiate a HaltErr:Fn action, which includes transitioning MMIO_STS.fn_gsv to the GSV_ERROR state. Descriptor errors including aborts alone do not drive a transition to the GSV_ERROR state.

Writing GSRV_STOP_HD to MMIO_CTL0.fn_gsr transitions MMIO_STS.fn_gsv to the GSV_STOPG_HD state. Writing any other value to MMIO_CTL0.fn_gsr is ignored.

4.1.5 G4: GSV_STOPG_HD ("Hard Stopping") State

In the GSV_STOPG_HD state, the function stops processing new descriptors and waits less patiently for all outstanding descriptors to complete; and aborts outstanding descriptors more aggressively to speed up the completion wait. The function shall log errors as necessary for each aborted descriptor. See *"4.3.5, Function and Context Stop Actions"*. Used as a last resort to stop the function.

When all contexts have reached their final state, and all previously received MMIO doorbells are fully evaluated (i.e. there are no outstanding data structure reads due to a prior doorbell), the function transitions to the GSV_STOP state.

While in this state, if an error occurs that prevents one or more contexts from stopping, the function shall initiate a HaltErr:Fn action, which includes transitioning MMIO_STS0.fn_gsv to the GSV_ERROR state.

Writing any value to MMIO_CTL0.fn_gsr is ignored.

4.1.6 G5: GSV_ERROR State

If an uncorrectable error prevents further safe operation of a function across all of its associated contexts, the function shall initiate a HaltErr:Fn action, which includes transitioning MMIO_STS0.fn_gsv to the GSV_ERROR state. If MMIO_CTL0.fn_err_intr_en is "1", an interrupt to software is generated upon entering this state.

All DMA operations from remote functions targeting this function's local address spaces are aborted in this state. It is implementation-dependent if the function cleans up any outstanding DMA operations in this state.

It is implementation dependent if the function performs any context-specific state changes or stop actions when it enters this state; software shall not rely upon it. Software may return the function to the GSV_STOP state by performing one these actions.

1. Set MMIO_CTL0.fn_gsr to GSRV_RESET and wait until MMIO_STS0.fn_gsv is GSV_STOP. It is implementation dependent what the contents of context-specific state are after a GSRV_RESET is requested.
2. Perform a function-level reset at the device level.

Writing any value other than GSRV_RESET to MMIO_CTL0.fn_gsr is ignored.

4.1.7 Function Reset and Outstanding DMA Requests

When a device-level reset is being performed on an SDXI function, there may be outstanding DMA requests from this specific function to both local and remote address spaces. This function may also receive DMA from other SDXI functions targeting this function's local address spaces. Those remote functions may continue to issue new DMA requests attempting to access this function's local address spaces while this function is resetting.

An SDXI function shall terminate all processing (descriptor ingest, descriptor execution, incoming RKey-based access, error reporting, etc.) and expeditiously clear (complete or cleanly abort) any outstanding external interactions. When complete, the function shall be stopped and MMIO_STS0.fn_gsv shall equal GSV_STOP.

It is privileged software's responsibility to notify the remote software contexts that the target function is being reset. It shall also ensure either orderly or unorderly removal of all associated connections prior to re-enabling the function that was reset.

4.1.8 Activation of the SDXI Function by Software

Software shall activate the SDXI function from the GSV_STOP state by performing the following steps.

1. Configure Capabilities:
    a. Read MMIO_CAP0 and MMIO_CAP1 to discover the supported SDXI features. If restoring saved function content, verify that the function's capabilities match the requirements of the saved function content.
    b. Write MMIO_CTL2 to configure the SDXI features set exposed to driver software.
2. Context Level 2 Table Setup:
    a. Allocate an aligned 4KB region of memory for the Context Level 2 Table
    b. Initialize all Context Level 2 Table entries to "0" or restore them from the saved function content and adjust accordingly.
    c. Program the MMIO_CXT_L2 register to point to the Context Level 2 Table location in memory.
        i. If guest virtual addressing is required, set MMIO_CTL0.fn_pasid and MMIO_CTL0.fn_pasid_vl to appropriate values.
3. Context Level 1 Table Setup:
    a. Allocate an aligned 4KB region of memory for the first Context Level 1 Table (contexts 0 to 127). Other Context Level 1 Tables may also be allocated at this step.
    b. Initialize these Context Level 1 Table entries to "0" or restore them the saved function content and adjust accordingly.
    c. Initialize the Context level 2 table to point to the Context Level 1 tables allocated above.
4. Administrative Context:
    a. Recommended: if not restoring a saved administrative context, software should create the function's administrative context (context 0) and all associated structures.
    b. The context's CXT_STS.state value should be set to "CXTV_RUN" to allow the context to be jump started, see step 10b.
5. Mailbox: If present, initialize the mailbox registers
6. If restoring saved function and context state, restore and adjust the state as appropriate.
7. Error Log: Initialize or restore the Error Log. Refer to *"3.4.2, Error Log Initialization"* for more details.
8. Software may also need to configure and enable additional PCIe standards-based features such as enabling PCIe "AtomicOp Requester Enable" (PCIe Device Control 2 Register), MSI/MSI-X, ATS, PRI, and TPH based on the desired use cases.
9. Set MMIO_CTL0.fn_gsr to GSRV_ACTIVE.
    a. Wait until MMIO_STS0.fn_gsv is GSV_ACTIVE or GSV_ERROR before performing any other actions on the function.
    b. If the function state becomes GSV_ERROR, software shall use the procedures described in *"4.1.6, G5: GSV_ERROR State"* to transition the function to GSV_STOP.
10. Once the function is at GSV_ACTIVE state, software may start the created and restored contexts using one of the methods below.

a. From the PF, the contexts of a VF can be started using a P2V.DSC_CXT_START_RS operation.

b. From the local function, context 0 can be "jump started" (See *"4.3.4, Starting A Context and Context Signaling"*); then a LOC.DSC_CXT_START_RS operation can be issued from that context to start the remaining ones.

4.1.9 Stopping of the SDXI Function by Software

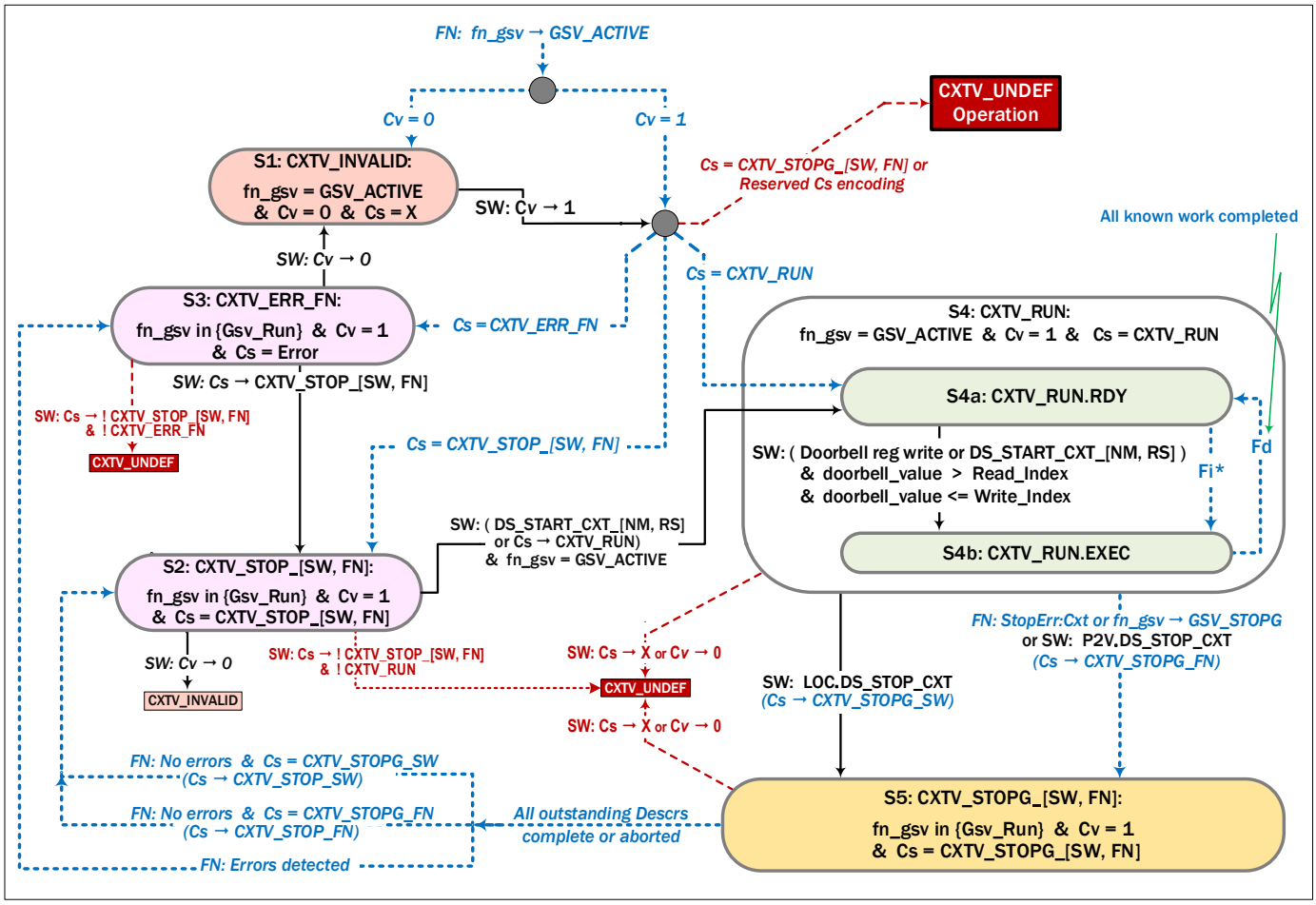
Software shall transition the SDXI function from GSV_ACTIVE to GSV_STOP state by performing the following steps.

1. Set MMIO_CTL0.fn_gsr to GSRV_STOP_[SF, HD].
   a. This will cause the function to enter GSV_STOPG_[SF, HD] and prevent any new operations from being processed in all contexts of the function.
2. Wait until MMIO_STS0.fn_gsv is GSV_STOP or GSV_ERROR.
   a. If the function is at GSV_STOPG_SF and is taking longer than acceptable to stop, software can Set MMIO_CTL0.fn_gsr to GSRV_STOP_HD to change the completion policy to hard-stop; note, this may cause more outstanding descriptors to be aborted.
3. If the function state becomes GSV_ERROR, software shall use the procedures described in *"4.1.6, G5: GSV_ERROR State"* to transition the function to GSV_STOP.
4. Once the function is at GSV_STOP, software may save the set of all current function contents including the error log, and optionally restore a different set of function contents.

4.2 SDXI Context State

An SDXI context operates serially within a number of basic states determined by the CXT_STS.state location in memory, the context valid bits (CXT_L2_ENT.vl, CXT_L1_ENT.vl, and CXT_CTL.vl), and the function state.

Figure 4-2: SDXI Context States and Basic State Transitions



CXT_STS.state	
0b0000	CXTV_STOP_SW
0b0001	CXTV_RUN
0b0010	CXTV_STOPG_SW
0b0100	CXTV_STOP_FN
0b0110	CXTV_STOPG_FN
0b1111	CXTV_ERR_FN

Key in {Set}	The value of "key" is within the values belonging to "Set"
{Gsv_Run}	The set containing the values GSV_ACTIVE and GSV_STOPG_[SF, HD]
Cs	CXT_STS.state
fn_gsv	Target Function's MMIO_STS0.fn_gsv
----SW: ----	Software Initiated State Transition
----FN:----	Function Initiated State Transition
----nnn----	Software Initated State Transition causing CXTV_UNDEF.

!value	For booleans, "!1" = "0" and "!0" = "1". For non-booleans, maps to any other allowed attribute value that is not "value".
key = value	Indicates that a state attribute ("key") is "value". When used in a state transition, the statement must be true.
key → value	A transition of a state attribute ("key") to "value".
Cv	Aggregate of (CXT_L2_ENT.vl and CXT_L1_ENT.vl and CXT_CTL.vl). Cv = 1 means all valid bits are "1". Cv = 0 means at least one valid bit is "0". Cv → n, means one or more valid bits have been changed such that Cv = n.
Fi*	Implementations may transition into CXTV_RUN.EXEC state without receiving a doorbell write by reading Write_Index and Read_Index from memory. Regardless, Software shall always issue doorbell writes in order to ensure correct operation.
Not shown: Regardless if an Administrative Descriptor is issued by the PF or the Local function (target), the target function's Gsv is used in determining its state transitions	

4.2.1 S1: CXTV_INVALID State

The SDXI function is in GSV_ACTIVE state and managing the context; but the context is invalid because one or more of the following attributes are not "1": CXT_L2_ENT.vl, CXT_L1_ENT.vl, and CXT_CTL.vl. In this state, the function does not process descriptors and does not respond to doorbell writes for the context. This state can be entered from the following.

- CXTV_STOP_[SW, FN] or CXTV_ERR_FN when software changes one or more of the context valid bits to "0".
- When the function transitions to GSV_ACTIVE and one or more of the context valid bits are "0".

This state can transition to the following.

- CXTV_RUN, CXTV_ERR_FN, or CXTV_STOP_[SW, FN] based on the value of CXT_STS.state when the function is at GSV_ACTIVE and software changes one or more of the context valid bits such that all are at "1".
- CXTV_UNDEF when the function is at GSV_ACTIVE, software changes one or more of the context valid bits such that all are at "1", and CXT_STS.state is not CXTV_RUN, CXTV_ERR_FN, or CXTV_STOP_[SW, FN].

Unless explicitly required by an action or operation to suppress or signal an error, an implementation of an SDXI function should suppress signaling errors on a context in CXT_INVALID state. However, the implementation may not be able to suppress the signaling of these errors in all cases. As a result, software shall expect and dispose of unexpected errors on a context in CXT_INVALID state.

4.2.2 S2: CXTV_STOP_[SW, FN] States

In these states the function does not process descriptors and does not respond to doorbell writes for the context. The target context shall be stopped at a descriptor boundary. This state can be entered from the following.

- CXTV_STOPG_[SW, FN] by the function when it detects that no error occurred in context processing and context execution.
- CXTV_ERR_FN when software changes CXT_STS.state to CXTV_STOP_[SW, FN]. (It is recommended that software resolve the error before making this change.)
- CXTV_INVALID when the function is at GSV_ACTIVE, CXT_STS.state is CXTV_STOP_[SW, FN], and all context valid bits are "1".

If context 0 (Administrative context) enters this state, the function shall invalidate and exclude all privately cached copies of LVL_L2 data associated with context 0 until the context is once again at CXTV_RUN. (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching"*)

This state can transition to the following.

- CXTV_RUN when the function is at GSV_ACTIVE and software performs either a successful DSC_CXT_START_[NM, RS] operation or a change of CXT_STS.state to "CXTV_RUN (see *"4.3.4, Starting A Context and Context Signaling"*).
- CXTV_INVALID when software changes one or more of the context valid bits to "0".
- CXTV_UNDEF when software changes CXT_STS.state to values other than CXTV_STOP_[SW, FN] and CXTV_RUN.

4.2.2.1 CXTV_STOP_SW State

The context was stopped by software using a local-function DSC_CXT_STOP operation on the context.

4.2.2.2 CXTV_STOP_FN State

The context was stopped either by a transition of the function state from GSV_ACTIVE or a P2V.DSC_CXT_STOP operation.

4.2.3 S3: CXTV_ERR_FN

This is the context error state; the function does not process descriptors and does not respond to doorbell writes for the context. The target context shall be stopped at a descriptor boundary. In response, software shall diagnose and correct any underlying error conditions. Once the error has been resolved, software may transition the target context to CXTV_STOP_[SW, FN]. This state can be entered from the following.

- CXTV_STOPG_[SW, FN] when the function detects an error in context processing or context execution.
- CXTV_INVALID when the function is at GSV_ACTIVE, CXT_STS.state is CXTV_ERR_FN, and all context valid bits are "1".

If context 0 (Administrative context) enters this state, the function shall invalidate and exclude all privately cached copies of LVL_L2 data associated with context 0 until the context is once again at CXTV_RUN. (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching"*)

This state can transition to the following.

- CXTV_STOP_[SW, FN] when software changes CXT_STS.state to CXTV_STOP_[SW, FN].
- CXTV_INVALID when software changes one or more of the context valid bits to "0".
- CXTV_UNDEF when software changes CXT_STS.state to values other than CXTV_ERR_FN and CXTV_STOP_[SW, FN].

4.2.4 S4: CXTV_RUN.[RDY, EXEC] States

In this state, the context has been enabled to process descriptors and responds to doorbell writes. This state can be entered from the following.

- CXTV_STOP_[SW, FN] when the function is at GSV_ACTIVE and software performs either a successful DSC_CXT_START_[NM, RS] operation or a change of CXT_STS.state to CXTV_RUN.
- CXTV_INVALID when the function is at GSV_ACTIVE, CXT_STS.state is CXTV_RUN, and all context valid bits are "1".

Entry to this state always enters the CXTV_RUN.RDY substate.

The CXTV_RUN state can transition to the following.

- CXTV_STOPG_FN when the function detects an error in context processing or context execution.
- CXTV_STOPG_FN when the function transitions to GSV_STOPG.
- CXTV_STOPG_FN by a software-initiated, PF-to-VF-function, administrative DSC_CXT_STOP operation.
- CXTV_STOPG_SW by a software-initiated, local-function, administrative DSC_CXT_STOP operation.
- CXTV_UNDEF when software changes CXT_STS.state to any value or one or more of the context valid bits to "0".

4.2.4.1 S4a: CXTV_RUN.RDY State

In this substate of CXTV_RUN, the context is enabled ("ready") to process descriptors, all previous context descriptors have been completed, and the function is evaluating whether new descriptors are available. The context may remain in this state until the function determines that there are available descriptors.

The SDXI function may transition the context to the CXTV_RUN.EXEC (S4b) state to process available descriptors when the following conditions are true.

1. The context is in CXTV_RUN.RDY.
2. The context's Doorbell register is updated externally or internally with a doorbell_value as described in *"4.3.3, Doorbell Register and Context Signaling"*
3. The doorbell_value is greater than Read_Index and the doorbell_value is less than or equal to Write_Index.

4.2.4.2 S4b: CXTV_RUN.EXEC State

In this substate, the context is actively processing new descriptors. The function may return the context to CXTV_RUN.RDY state when the last known doorbell_value equals Write_Index.

4.2.5 S5: CXTV_STOPG_[SW, FN] States

In these states, the function stops processing new descriptors for the context and waits for outstanding context descriptors to complete based on the completion wait policy.

- **Soft-Stop (SF) Wait**: The function shall wait until all outstanding descriptors complete naturally or until an implementation-defined timeout period has expired at which point any remaining descriptors are aborted.
- **Hard-Stop (HD) Wait**: The function waits less patiently for all outstanding descriptors to complete; and may abort outstanding descriptors more aggressively to speed up the completion wait. This policy can be applied by an initial stop action on the context or by later stop actions on the same context. Once applied, the completion policy cannot be relaxed while in the context is in this state.

Refer to CXTV_STOPG_SW and CXTV_STOPG_FN for more details.

4.2.5.1 CXTV_STOPG_SW State

This state can be entered from the following.

- CXTV_RUN by a software-initiated, local-function, administrative DSC_CXT_STOP operation.

This state can transition to the following.

- CXTV_STOP_SW by the function when it detects that no error occurred in context processing and context execution.
- CXTV_ERR_FN when the function detects an error in context processing or context execution.
- CXTV_UNDEF when software changes CXT_STS.state to any value or one or more of the context valid bits to "0".

4.2.5.2 CXTV_STOPG_FN State

This state can be entered from the following.

- CXTV_RUN when the function detects an error in context processing or context execution.
- CXTV_RUN when the function transitions to GSV_STOPG.
- CXTV_RUN by a software-initiated, PF-to-VF-function, administrative DSC_CXT_STOP operation.

This state can transition to the following.

- CXTV_STOP_FN by the function when it detects that no error occurred in context processing and context execution.
- CXTV_ERR_FN when the function detects an error in context processing or context execution.
- CXTV_UNDEF when software changes CXT_STS.state to any value or one or more of the context valid bits to "0".

4.2.6 Context-Undefined Operation (CXTV_UNDEF)

When privileged software directly and illegally modifies a context's CXT_STS.state or context valid bits (CXT_L2_ENT.vl, CXT_L1_ENT.vl, and CXT_CTL.vl) in memory, the context operation is architecturally undefined. Because the SDXI function may cache context control state and may still be executing outstanding operations issued before the illegal modification, the inconsistent or stale context control state may result in unexpected context operation. This can include the use of stale data, logged context errors, and an unexpected change of CXT_STS.state due to context errors or previous outstanding administrative operations on the context.

Note that the effects of such undefined context operation are limited to only the results that may be produced by any valid context state. Furthermore, no other context nor the SDXI function shall be affected when a particular context has undefined operation.

Although correctly written privileged software shall never cause context-undefined operation, there is no method for software to detect it after the fact. In addition, for the benefit of performant SDXI implementations, the SDXI architecture does not require nor recommend that an SDXI function attempt to detect context-undefined operation caused by incorrect privileged software.

Context-undefined operation is exited when the context is stopped by any operation including DSC_CXT_STOP operations, function stop operations, and StopErr:Cxt (context or descriptor error actions).

4.3 Function and Context Operations

4.3.1 SDXI Memory-Based Data-Structure Hierarchy and Caching

SDXI-defined memory-based data structures are organized into a hierarchy of levels for the purposes of function and software management. The validity of a data structure is based on its explicit "valid (vl)" field in memory, if present, and its inherited validity derived from the validity of all higher levels in the hierarchy that point to it. A structure lacks inherited validity if any of these higher levels are invalid. (For example, a context level 1 entry can only be considered valid if its explicit CXT_L1_ENT.vl is "1", its associated higher level CXT_L2_ENT.vl is "1", and the SDXI function is at GSV_ACTIVE or GSV_STOPG_[SF, HD].)

The SDXI function may cache these memory-based data structures using private caches that are not coherent with respect to processor caches. Data in both the valid and invalid states may be cached in this manner across all contexts within the function.

The memory-based data-structure hierarchy (smallest to largest) and its suitability for caching is shown below.

1. **Lowest-Level, Non-Cached Data:** the Write_Index value and descriptors not located between Read_Index and Write_Index in the context descriptor ring. The validity of these structures is inherited from the validity of their associated CXT_CTL.
2. **Lowest Level, Cached Data:** May be cached by the SDXI implementation.
    a. **LVL_CXT_STS**: Single Context Status (CXT_STS) entries and their fields. May be cached only when CXT_STS.state is CXTV_RUN or CXTV_STOPG_[SW, FN]. The validity of the structure is inherited from the validity of its CXT_CTL.
    b. **LVL_AKEY**: Single AKey Table Entries. The validity of this structure is derived from its explicit "valid" field and the validity of its associated CXT_L1_ENT.
    c. **LVL_RKEY**: Single RKey Table Entries. The validity of this structure is derived from its explicit "valid" field and the validity of LVL_FN.
    d. **LVL_DESCR:** descriptors with the valid (vl) field set to "1" that are located between Read_Index and Write_Index. The validity of such descriptors is further qualified by the validity of the associated CXT_CTL.
3. **LVL_CXT_CTL**: Single Context Control (CXT_CTL) entries and all subsidiary structures. This includes LVL_CXT_STS and LVL_DESCR. These structures may be cached. The validity of this level is derived from the explicit CXT_CTL.vl field and the validity of its associated CXT_L1_ENT.
4. **LVL_L1**: Single CXT_L1_ENT entries and all subsidiary structures. This includes the LVL_CXT_CTL and all LVL_AKEY associated with the CXT_L1_ENT. These structures may be cached. The validity of this level is derived from the explicit CXT_L1_ENT.vl field and the validity of its associated CXT_L2_ENT.
5. **LVL_L2**: Single CXT_L2_ENT entries and all subsidiary structures. This includes all LVL_L1 associated with the CXT_L2_ENT. These structures may be cached. The validity of this level is derived from the explicit CXT_L2_ENT.vl field and the validity of its associated LVL_FN.
6. **LVL_FN**: All memory-based fields of the function. This includes all LVL_L2 and LVL_RKEY. These structures may be cached as described above. Explicitly valid when the SDXI function is at GSV_ACTIVE or GSV_STOPG_[SF, HD].

4.3.1.1 SDXI Update Operations for Modified Data Structures

When software has modified a memory-based data structure that may be privately cached by the function, software shall appropriately signal the function to update its internal copies of that data structure, if any. The SDXI signaling operations and mechanisms used by software for this purpose are described in the list below. The software procedures using these operations are described in *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*"

1. DSC_UPD_[CXT, AKEY, RKEY, FN] descriptor operations. These signal the function to start the updating of all internal copies of a memory-based data structure and its subsidiary structures. The operations are scoped to match the hierarchy of memory-based data-structures described in *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*".
    a. The SDXI implementation may perform the update by re-acquiring a copy of the data from system memory or by simply invalidating all internal copies of the data in its private caches. Software shall not rely upon how the update is performed.
    b. Note that a successful completion of a DSC_UPD operation only indicates that background update actions shall be started; it does not indicate that the update actions have completed nor that their effects are visible. Software shall use the DSC_SYNC operation to ensure that the background update actions have completed.
2. DSC_SYNC.[CXT, AKEY, RKEY, FN] descriptor operation. This operation synchronizes with the completion of the appropriately scoped background update actions initiated by previous DSC_UPD operations. Once the function completes a DSC_SYNC operation, the function shall have performed all associated updates.
3. Context Stop. Whenever a context is stopped, the SDXI function shall invalidate all internal copies of the context's CXT_STS and ensure that the function excludes re-acquiring a copy of CXT_STS until the context is once again at CXTV_RUN.
4. A local DSC_CXT_STOP operation with cxt_start = 0 and cxt_end = 0 shall invalidate all LVL_L2 data associated with context 0 and ensure that the function excludes re-acquiring a copy of the LVL_L2 data until context 0 is once again at CXTV_RUN.

Note, that for any requested level of the memory-based data-structure hierarchy to update, an SDXI implementation may update (or invalidate and exclude) any associated higher level of the hierarchy.

4.3.1.2 Software Procedure For Modifying Memory-Based Data Structures

When software modifies a memory-based data structure that can be privately cached by the SDXI function, it shall follow the procedures described in this section to ensure that the modification is performed correctly.

Procedure For Modifying A Memory-Based Data Structure:

1. If the data structure is not allowed to be cached by the SDXI function, software may modify it without restriction and no additional steps are required.
2. If a data structure does not currently possess inherited validity, software may modify it without restriction. Software then proceeds to step 5 in this flow.
    a. If a hierarchy of structures are being transitioned from invalid to valid, software should update the hierarchy from the smallest scope to the largest to benefit from this fact.
3. If a data structure with an explicit valid field has inherited validity and the modification will transition the explicit field from "invalid" to "valid", software shall ensure that only the last (terminal) write to the structure causes the transition. Furthermore, software shall ensure that all earlier writes to the structure, if any, are made globally visible before such a terminal write. Software then proceeds to step 5 in this flow.
    a. Example: for a valid context, transitioning an unused AKEY entry from invalid to valid.
4. The following steps apply to a data structure that is considered valid -- it has inherited validity and its explicit valid field, if present, is true. (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*").
    a. If the data structure is encompassed by LVL_CXT_CTL and the CXT_CTL is explicitly valid, software shall only modify the structure when the associated CXT_STS.state is CXTV_STOP_[SW, FN] or CXTV_ERR_FN. Software then proceeds to step 5 in this flow.
        i. Example: modifying CXT_CTL.cxt_sts_ptr for a valid context.
    b. For all other data structures, if the-structure lacks an explicit valid field, software shall only modify it after software has properly invalidated the data structure that points to it (the hierarchy level above it). Software then proceeds to step 5 in this flow.
        i. Example: modifying CXT_L1_ENT.cxt_ctl_ptr for a valid context requires that CXT_L1_ENT.vl be set to invalid.
    c. For all other data structures, if the structure has an explicit valid field set to "valid", and software ensures that the structure is not actively in use by any context nor the SDXI function, software shall ensure that the first write to the structure transitions it to an invalid state before further modification. Furthermore, software shall ensure that this write shall be made globally visible before any subsequent write to the structure. Software then proceeds to step 5 in this flow. Further notes below.
        i. A software mechanism must exist for how software ensures that a data structure outside of LVL_CXT_CTL is not in use. That mechanism is outside the scope of this specification.
        ii. Example: invalidating a valid CXT_L1_ENT requires that software ensure that it is not in use before the invalidation.
        iii. Example: invalidating a valid AKEY_ENT requires that software ensure that it is not in use before the invalidation.

Update and Synchronization Of A Modified Memory-Based Data Structure:

5. Software shall issue a DSC_UPD operation targeting the modified data-structure hierarchy level. Software may target a higher level for the update, but shall never target a lower level below the actual modified data structure.
    a. Example: For a valid CXT_L2_ENT, modifying a member CXT_L1_ENT requires software to issue at least a DSC_CXT_UPD.L1 operation, but DSC_CXT_UPD.L2 is also allowed.
    b. Example: For a valid CXT_L1_ENT, modifying a member AKEY_ENT requires software to issue at least a DSC_AKEY_UPD operation, but DSC_CXT_UPD.L1 and DSC_CXT_UPD.L2 are also allowed.
6. Software shall then issue a DSC_SYNC operation targeting the modified data-structure hierarchy level and wait for its completion. Software may target a higher level for the update but, shall never target a lower level below the actual modified data structure.
    a. Example: For a valid CXT_L2_ENT, modifying a member CXT_L1_ENT requires software to issue at least a DSC_SYNC.CXT operation, but DSC_SYNC.FN is also allowed.
    b. Example: For a valid CXT_L1_ENT, modifying a member AKEY_ENT requires software to issue at least a DSC_SYNC.AKEY operation, but DSC_SYNC.CXT and DSC_SYNC.FN are also allowed.

4.3.2 Check Valid Context

When an SDXI function is starting or stopping a context, the function shall check that the key memory data structures of the context are read accessible, and enabled or valid. This check is referred to as the "ChkValid:Cxt" action. When the check fails in the flow below, no further steps in the flow are performed, and the fail signature returned shall be either LogErr:Cxt or Invalid:Cxt (see "3.4, Error Log" for details.)

1. The function ensures the following; if any are not true, then the check fails with LogErr:Cxt and these set of steps are exited.
    a. MMIO_CTL2.max_cxt is less than or equal to MMIO_CAP1.max_cxt.
    b. The context number is less than or equal to MMIO_CTL2.max_cxt.
2. The function shall ensure that it is not using stale values of the following context structures:
    a. Context Status (which includes ContextState and Read_Index) and Write_Index.
3. The function shall access and validate the following list of structures. The function shall use naturally-aligned atomic accesses to read these structures from memory if not cached. If the function cannot access a specified structure, then the check fails with LogErr:Cxt and these set of steps are exited.
    a. Access the valid bit for the context level 2 table entry (CXT_L2_ENT.vl) that corresponds to the context number; if CXT_L2_ENT.vl is "0" then the check fails with Invalid:Cxt and these set of steps are exited.
    b. If CXT_L2_ENT.vl is "1", access CXT_L2_ENT.Lvl_1_ptr and use it to access the valid bit for the context level 1 table entry (CXT_L1_ENT.vl) that corresponds to the context number. If CXT_L1_ENT.vl is "0" then the check fails with Invalid:Cxt and these set of steps are exited.
    c. If CXT_L1_ENT.vl is "1", access CXT_L1_ENT.pv, CXT_L1_ENT.cxt_pasid, and CXT_L1_ENT.cxt_ctl_ptr. Use these to access the valid bit for the Context Control (CXT_CTL.vl). If CXT_CTL.vl is "0" then the check fails with Invalid:Cxt and these set of steps are exited.
    d. If CXT_CTL.vl is "1", verify access to these memory locations:
        i. The starting memory address pointed to by CXT_CTL.ds_ring_ptr,
        ii. The starting memory address pointed to by CXT_CTL.cxt_sts_ptr,
        iii. The starting memory address pointed to by write_index_ptr.
        iv. If CXT_CTL.vl is "1", read CXT_STS.state; if it is a reserved encoding then the check fails with LogErr:Cxt and these set of steps are exited.

If the check fails, the function shall invalidate and exclude the caching of the context's Context Status (CXT_STS) and take other operation-specific actions as required.

4.3.3 Doorbell Register and Context Signaling

As new descriptors are written to a descriptor ring, software updates the context's Write_Index location in system memory as appropriate. Because the SDXI function operates asynchronously with respect to a context and the Write_Index location is in system memory, software shall directly signal the function after updating Write_Index and the descriptor ring to ensure new descriptors are evaluated. Software signals the function by ensuring the update of the final Write_Index value, the doorbell_value, to a per-context, MMIO-mapped, architectural Doorbell register when the context is in CXTV_RUN. (See *"9.7, Doorbell Sections and Registers".*) The Doorbell register may be updated by software by the following mechanisms.

- Externally updated when the context is in CXTV_RUN and software writes a doorbell_value to the context's Doorbell register.
- Internally updated when the context is transitioned to the CXTV_RUN state by software issuing for the target context a DSC_CXT_START_[NM, RS] operation that specifies a doorbell_value.

The Doorbell register may also be internally updated by the function when the context is in CXTV_RUN and an SDXI implementation optionally chooses to read Write_Index and Read_Index from memory. Regardless of the SDXI implementation, software shall always issue Doorbell register writes when updating Write_Index in order to ensure correct operation.

The SDXI function evaluates a doorbell_value by the following ordered steps.

1. If the context is not in CXTV_RUN state at the time the Doorbell register is implicitly or explicitly written, the doorbell_value is ignored and the function logs no error and takes no action based on it; no further steps are done. Otherwise, proceed to the next step.
2. If the doorbell_value is supplied by a DSC_CXT_START_[NM, RS] operation and the value is 0xFFFF_FFFF_FFFF_FFFF, then the function shall discard this value and fetch the doorbell_value from the context's Write Index location.
3. If the doorbell_value is supplied by any other method, then it is implementation-dependent if the function uses the supplied doorbell_value or discards it and fetches doorbell_value from the context's Write Index location.
4. If the doorbell value is less than or equal to previous valid doorbell_value values received since the context was started, the function ignores the doorbell_value and takes no action. (This determination is reset every time the context is started.)
5. If the doorbell_value is less than or equal to the context's Read_Index value, it is implementation-dependent whether the function accesses the context's descriptor ring.
6. If the doorbell_value is greater than the context's Read_Index value, the function will examine the context's descriptor ring starting at Read_Index for descriptors to execute.
7. It is implementation-dependent whether the function uses cached values of context data structures in determining the location of the Read_Index and Write_Index locations in memory.
8. It is implementation-dependent whether the function logs an error if any part of the context needed to access the Read_Index and Write_Index locations is invalid or inaccessible.

There is no time requirement as to when the function acts on a supplied doorbell_value supplied by the above methods.

Software shall make no assumptions as to the implementation of the Doorbell register.

4.3.4 Starting A Context and Context Signaling

A context must be in the CXTV_RUN state for an SDXI function to evaluate it for new descriptors to execute. Furthermore, because of the asynchronous nature of an SDXI function, the function must be signaled when a CXTV_RUN context has new descriptors to execute. Each of these is an independent mechanism.

- Starting a Context: this is the mechanism of transitioning a valid context's CXT_STS.state to CXTV_RUN. Starting a context does not ensure that the function evaluates it for new descriptors to execute.
- Context Signaling: this is the mechanism of signaling the function that a context should be evaluated for new descriptors to execute using the specified doorbell_value. The signaling does not imply that the context is valid or at CXTV_RUN; the SDXI function will determine this independently and take appropriate action.

Building on the above concepts, there are several methods for software to start a target context within a target function and ensure that new descriptors are evaluated thereafter in a timely manner. Each of these may be combined with context signaling to ensure that new descriptors are evaluated by the function

1. Software issues a DSC_CXT_START_[NM, RS] operation from the function's administrative context to start a context in the same function using an appropriate doorbell_value. (See *"6.6.3, AdminGrp DSC_CXT_START_[NM, RS] Operations"*) This is the preferred method for all contexts except context 0, the administrative context.
2. From the PF, software issues a DSC_CXT_START_[NM, RS] operation targeting another function's context with an appropriate doorbell_value. This method is intended primarily for hypervisor usage.
3. "Jump Start 1": set the target context's CXT_STS.state in memory to CXTV_RUN when in the appropriate state (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*") and then write an appropriate doorbell_value to the context's Doorbell register. (See *"4.3.3, Doorbell Register and Context Signaling"*) This is the recommended way within a function to start context 0, the administrative context.

Using the DSC_CXT_START_[NM, RS] operation ensures that the function is not using stale values of key context structures. (See *"6.6.3, AdminGrp DSC_CXT_START_[NM, RS] Operations"*)

Using a "Jump Start" method requires that software explicitly ensure there are no in-flight doorbell writes before changing CXT_STS.state to CXTV_RUN, and that there is no stale context data before writing the doorbell register. Stale context data for a valid context may be avoided by using DSC_UPD and DSC_SYNC operations against the function or specific context (see *"6.6.9, AdminGrp DSC_SYNC Operation"*). In the case of context 0, stopping the SDXI function ensures stale context and in-flight doorbells are flushed (see *"3.5, Administrative Context (Context 0)"*). When a function transitions from GSV_STOP to GSV_ACTIVE state, the function shall have no stale context data cached for that context.

Software shall ensure that the function's context 0, the administrative context, is started before operating on other function contexts.

4.3.5 Function and Context Stop Actions

An SDXI function uses context stop actions to stop processing of new descriptors and to wait for previously issued descriptors to complete for targeted contexts. A stop action shall complete in a timely manner so that further administrative operations on the targeted contexts may be performed. The following SDXI context-stop actions are defined below.

1. **LOC.DSC_CXT_STOP.[SF, HD]:** This is a software-initiated, local-function, administrative DSC_CXT_STOP operation. The operation's Hard Stop (HS) bit indicates "SF" when HS = 0, and "HD" when HS = 1. The function shall invalidate and exclude the caching of the Context Status (LVL_CXT_STS). See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*".
2. **P2V.DSC_CXT_STOP.[SF, HD]**: This is a software-initiated, PF-to-VF-function, administrative DSC_CXT_STOP operation. The operation's Hard Stop (HS) bit indicates "SF" when HS = 0, and "HD" when HS = 1. The function shall invalidate and exclude the caching of the Context Status (LVL_CXT_STS). See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*".
3. **StopErr:Cxt**: The function detects an error: in processing or executing a descriptor; or in context operation. In this case the function shall initiate a background context stop action (StopErr:Cxt) and log a descriptor error (LogErr:Cxt). If the error relates to a specific descriptor, the function shall signal an error in the descriptor's completion status block (Signal_Err_Csb). The function shall invalidate and exclude the caching of the Context Status (LVL_CXT_STS).
4. **Stop[SF, HD]:Fn** : The function performs a stop action in response to a software-initiated stop of the function. (Software initiates stopping the function by writing GSRV_STOP_[SF, HD] to MMIO_CTL0.fn_gsr.) The function shall invalidate all cached memory-based data (LVL_FN). The function state will change based on this action; it is discussed in *"4.1.1, G0: GSV_STOP State"*.

> **Note: HaltErr:Fn** is a function-initiated halt and abort of all function activity in response to a function-wide error; it is not a stop-action. See *"4.1.6, G5: GSV_ERROR State"* for more details.

Stop actions apply one of the below completion-wait policies to each target context.

- **Soft-Stop (SF) Wait**: The function shall wait until all outstanding descriptors complete naturally or until an implementation-defined timeout period has expired at which point any remaining descriptors are aborted. The DSC_CXT_STOP.SF, StopSF:Fn, and StopErr:Cxt actions apply this policy
- **Hard-Stop (HD) Wait:** The function waits less patiently for all outstanding descriptors to complete; and may abort outstanding descriptors more aggressively to speed up the completion wait. Used as a last resort to stop the context. The DSC_CXT_STOP.HD and StopHD:Fn actions apply this policy.

When an outstanding descriptor is aborted because of a context stop action: its completion status indicates an error; an error is logged; and CXT_STS.state shall be set to CXTV_ERR_FN.

Because of the asynchronous nature of the SDXI function, the function may be performing multiple background stop actions concurrently against the same context or across different contexts. Some examples follow.

- Software may initiate a subsequent stop operation with hard-stop completion if a previous stop operation with soft-stop completion is not making acceptable forward progress.
- All of the following can be concurrent on the same context: a StopErr:Cxt initiated by the function; a LOC.DSC_CXT_STOP operation submitted by software administering a VF; and a Stop:Fn action requested by a hypervisor managing the function.

Unless these actions are explicitly synchronized by software, the SDXI function enforces no ordering among them. However, for a single context, the function ensures the following.

- If the context is already at CXTV_ERR_FN or CXTV_STOP_[SW, FN], subsequent stop actions are ignored.
- If the context is already at CXTV_STOPG_[SW, FN], future stopping actions may only strengthen the completion wait policy to hard-stop.
- Only one stop action can transition a context from CXTV_RUN to CXTV_STOPG_[SW, FN].

For each context targeted by a given stop-action operation, the function executes the following ordered set of steps below; the steps are also shown in *"Figure 4-3: Context Stop Action Flow"*.

K0. The function records the requested completion-wait policy (req_wait) specified by the stop action.
   a. The function records req_wait as "SF", if the stop action is DSC_CXT_STOP.SF, StopSF:Fn, or StopErr:Cxt.
   b. The function records req_wait as "HD", if the stop action is DSC_CXT_STOP.HD or StopHD:Fn.

K1. The function shall perform a Check Valid Context action (ChkValid:Cxt) described in *"4.3.2, Check Valid Context"*.

K2. If the check fails, the function shall end this stop action on the context and take further action described below.
   a. If the stop action is Stop[SF, HD]:Fn, take no further action.
   b. If the stop action is a StopErr:Cxt, initiate a HaltErr:Fn -- because the function should not initiate a StopErr:Cxt on a context that fails ChkValid:Cxt.
   c. If the stop action is a DSC_CXT_STOP with a ChkValid:Cxt failure signature of LogErr:Cxt, then log the appropriate error.
   d. If the stop action is a DSC_CXT_STOP with a ChkValid:Cxt failure signature of Invalid:Cxt, take no further action.

K3. If the context is already at CXTV_STOP_[SW, FN] or CXTV_ERR_FN, the function shall end this stop action on the context.

K4. If the context is not yet at CXTV_STOPG_[SW, FN] (which means that another stop-action has not yet reached this context), the function shall proceed to step K6. Otherwise, the function shall continue to the following step.

K5. The function shall set the current completion-wait policy (curr_wait) to the requested completion-wait policy (req_wait) as described below, and end this stop action on the context.
   a. If req_wait is "HD" (hard completion wait), the function shall ensure that curr_wait is set to "HD".
   b. If req_wait is "SF" (soft completion wait), the function shall not change curr_wait. (Thus, once a hard completion wait policy is applied, it cannot be overridden.)

K6. At this point, only one stop action shall be operating on the context. The function shall atomically write a "stopping" value to CXT_STS.state as described below, and then continue to the following step.
   a. The function shall atomically write CXTV_STOPG_SW, if the stop action is a LOC.DSC_CXT_STOP.[SF, HD].
   b. The function shall atomically write CXTV_STOPG_FN, if the stop action is a P2V.DSC_CXT_STOP.[SF, HD], Stop[SF, HD]:Fn, or StopErr:Cxt.

K7. The function shall terminate context execution at a descriptor boundary and initialize the context's current completion-wait policy (curr_wait) to the requested completion-wait policy (req_wait).

K8. The function then performs the stopping of the context using the current completion-wait policy (curr_wait). While the function is performing this step, any change of the context's curr_wait to "HD" by other stop actions on the same context shall take effect during this step at an implementation-defined point.

K9. The function returns to step K8 until one of the following mutually exclusive conditions occur.

    a. The context is stopped: the function goes to step K10.

    b. The function detects an implementation failure in stopping the context: the function initiates a HaltErr:Fn action and shall then complete this stop action on the context.

K10. The function shall invalidate and exclude all internal copies of the following data structures and ensure that the function excludes re-acquiring a copy of these data structures until the context is once again at CXTV_RUN. If successful, the function goes to step K11. If unable to invalidate and exclude copies of the specified data structures due to an implementation error, the function initiates a HaltErr:Fn action and shall then complete this stop action on the context.

    a. The context's CXT_STS.

    b. The associated LVL_L2 for context 0.

K11. If the stop action is StopErr:Cxt or a context error was detected during the stopping of the context, the function goes to step K13.

K12. The function shall atomically write a "stopped" value to CXT_STS.state as described below, and then complete this stop action on the context.

    a. The function shall atomically write CXTV_STOP_SW, if the current CXT_STS.state is CXTV_STOPG_SW.

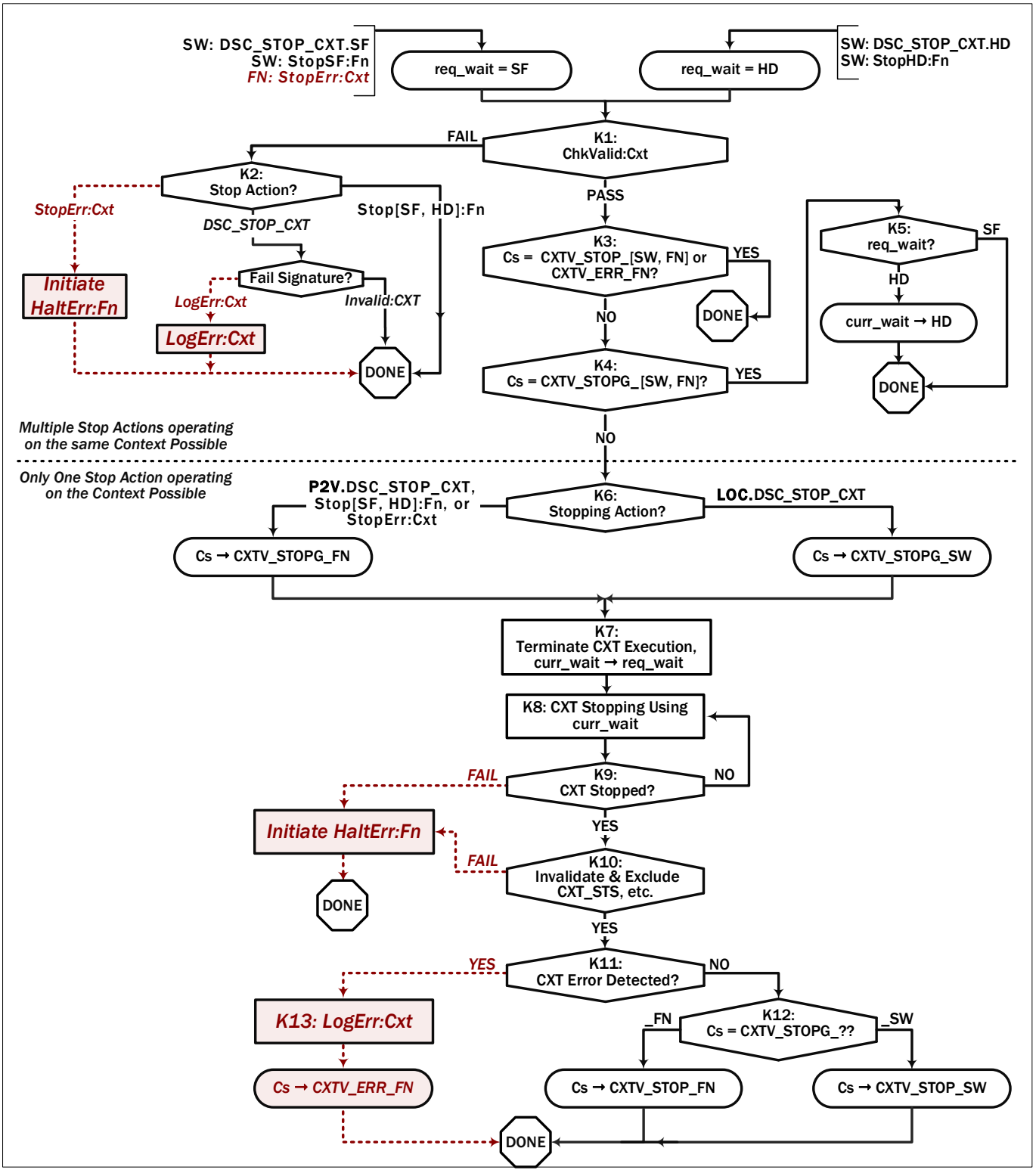
    b. The function shall atomically write CXTV_STOP_FN, if the current CXT_STS.state is CXTV_STOPG_FN.

K13. The function shall log the appropriate error (LogErr:Cxt) and then atomically write "CXTV_ERR_FN" to CXT_STS.state. The function shall then complete this stop action on the context.

Although not shown in the above per-context stop steps, when the function performs a Stop[SF, HD]:Fn action, it shall invalidate all privately-cached memory-based data (LVL_FN) when it completes the operation.

There may be a long latency from when a context stop action is initiated by privileged software to when the context is actually stopped by the function due to many descriptors that may still be in execution. If privileged software knows in advance that it will later stop the context, it may hint to the function earlier that it should start "draining" (stop executing new descriptors) the context by clearing the context's CXT_L1_ENT.ka (Keep Active Hint) to "0". This may reduce the time that software must wait for stopping the context later.

Figure 4-3: Context Stop Action Flow



4.3.6 Context Ring Submission Hint

When privileged software is momentarily stopping a context to change privileged state, it may want to hint to other software that new descriptors may still be submitted to a context during the interim. This hint may minimize the impact to software using the context while the context is stopping or stopped. To ensure a consistent software implementation, SDXI defines the ring submission hint (rsh) in the Context Status (CXT_STS.rsh). This bit is solely used by software and has no impact on the SDXI function, which never uses nor writes it.

The ring submission hint is intended to only be valid to software when the CXT_STS.state has a value of CXTV_STOPG_SW or CXTV_STOP_SW; it should be ignored otherwise by software. Software shall write CXT_STS.rsh with a byte-sized access only; this avoids conflicting writes by the function to CXT_STS.state, which is in a different byte.

- When CXT_STS.rsh = 1, software may continue to submit new descriptors to the context.
- When CXT_STS.rsh = 0, software is encouraged to not submit new descriptors. See the below table for intended operation.

Table 4–1: CXT_STS.rsh Mappings

CXT_STS.rsh (bit 8)	CXT_STS.state (bits 3:0)	Definition
0	0b0000	Context is in CXTV_STOP_SW more permanently. Software is encouraged to not submit new descriptors.
0	0b0010	Context is in CXTV_STOPG_SW more permanently. Software is encouraged to not submit new descriptors.
1	0b0000	Context is in CXTV_STOP_SW temporarily. Software may submit new descriptors during the interim.
1	0b0010	Context is in CXTV_STOPG_SW temporarily. Software may submit new descriptors during the interim.
X	0b0001	Context is in CXTV_RUN. Software may submit new descriptors.
X	Any Other Value	Software may not submit descriptors.

Software may use this procedure to determine if it should submit new descriptors for a context:

```
// Let csx = CXT_STS[8:0]
// Let rsh = csx & 0b100000000
// Let cse = csx & 0b1111

check0 = ( cse == CXTV_STOP_SW || cse == CXTV_STOPG_SW );

if ( rsh && check0 || cse == CXTV_RUN ){
      submit_work();
}
```

Privileged software may use many approaches to configuring CXT_STS.rsh. A simple approach is for privileged software to set CXT_STS.rsh to "1" before starting the context for normal usage and to leave it unchanged afterwards until the context is no longer available for normal usage. Thus, CXT_STS.rsh never needs to be changed during the small interims when privileged software is modifying privileged context state.

4.4 Atomic Operation Support

The usage of atomic operations by the SDXI function depends on both the SDXI function supporting those operations ("function-supported atomic operations") and the capability and configuration of the processor and platform interface connections to the SDXI function to support those operations ("interface-supported atomic operations"). Privileged software shall verify both before exposing an atomic operation to an SDXI context. The following section gives more detail.

An SDXI implementation may support none, some, or all of the following function-supported atomic operations; each set of operations has an enumeration bit in the SDXI MMIO register space.

1. Atomic accesses to the the completion status block (CST_BLK) associated with an SDXI descriptor operation; enumerated by MMIO_CAP0.cs_cap. See *"4.4.1, Completion-Status Capabilities"*.
2. The full set of descriptors in the atomic operations group; enumerated by MMIO_CAP1.opb_000_cap[3].
3. The minimal set of descriptors in the atomic operations group; enumerated by MMIO_CAP1.opb_000_cap[5].

An SDXI implementation may connect to its hosting hardware platform and other SDXI functions through one or more connection interfaces; examples include but are not limited to PCIe, CXL, and proprietary interfaces. Interface-supported atomic operations to and from the SDXI function may not exist or may not be configured. Platform and interface-specific mechanisms are required to determine if the capability is available and enabled. For example, see "*8.4, PCIe Atomic Capabilities Discovery and Enablement*" for details of how privileged software can determine this for an SDXI function using a PCIe interface.

Privileged software shall verify ***both*** of the following before exposing an atomic operation to an SDXI context: interface-supported atomic operations are supported and enabled between the SDXI function and any relevant target; both the SDXI function and any relevant target have the relevant function-supported atomic operations.

4.4.1 Completion-Status Capabilities

Each SDXI descriptor operation may specify a completion status block ("CST_BLK") whose fields are used by the SDXI function to update (modify) the descriptor operation's completion status and report relevant errors. The producer of the descriptor initializes the CST_BLK before submitting the descriptor. (See *"6.1.2, Completion Status Block"*.)

The SDXI function shall support modifying the CST_BLK with respect to other accesses either atomically or non-atomically; it may also support both capabilities and allow the producer to choose between the two using a descriptor control flag described in *"4.4.2, Completion-Status Modes"*. The SDXI function reports these capabilities in the MMIO_CAP0.cs_cap field as shown in the below table The details of the capabilities follow in the list below.

Table 4–2 Completion-Status Capabilities

MMIO_CAP0.cs_cap	Completion Status Capability
00b	Atomic completion-status capability supported.
01b	Reserved
10b	Both atomic and non-atomic completion-status capabilities are supported.
11b	Non-atomic Completion-Status capability supported.

1. Function: Atomic-Completion-Status Capability: the SDXI function supports atomic modification of the CST_BLK. when MMIO_CAP0.cs_cap is "00b" or "10b".
    a. The function shall perform modification of the CST_BLK and decrementing of the CST_BLK.signal field atomically with respect to other accesses of the same. The capability depends on interface-supported atomic operations.
    b. An atomic decrement of 0 in CST_BLK.signal results in 0xFFFF_FFFF_FFFF_FFFF.
    c. If the function only supports the atomic completion-status capability but does not have access to interface-supported atomics, then privileged software must not enable context execution for the SDXI function.

2. Function: Non-Atomic Completion-Status Capability: the SDXI function supports modification of the CST_BLK without using atomic operations when MMIO_CAP0.cs_cap is "10b" or "11b".
    a. The function shall perform the following without using atomic operations: modification of the CST_BLK; and transitioning the CST_BLK.signal to a terminal value of "0". The function shall not depend on interface-supported atomic operations to implement the capability. The function may not choose to evaluate the initial value of CST_BLK.signal.
    b. If the producer does not ensure that the initial value of CST_BLK.signal is "1", the function may return any terminal result in CST_BLK.signal; however, it will not generate an error.

4.4.2 Completion-Status Modes

Based on the completion-status capabilities described in section *"4.4.1,Completion-Status Capabilities"*, privileged software determines the related, but not identical, completion-status mode availability for each context. It exposes the availability of these modes through CXT_CTL.csa ("completion-status mode availability"); note, however, that this field is informational only and not enforced by the SDXI function. Privileged software shall also communicate and negotiate the available modes to the producer for a given context. This communication is outside the scope of the SDXI specification, but a suggested mechanism is shown in *"8.4, PCIe Atomic Capabilities Discovery and Enablement"*. The following discussion describes the availability modes and how they are used.

Table 4–3 Completion-Status Mode Availability

CXT_CTL.csa	Definition
0b	Both atomic completion-status mode and simple completion-status mode are available to the context.
1b	The context shall only use simple completion-status mode.

Table 4–4 Completion-Status Mode Requirement

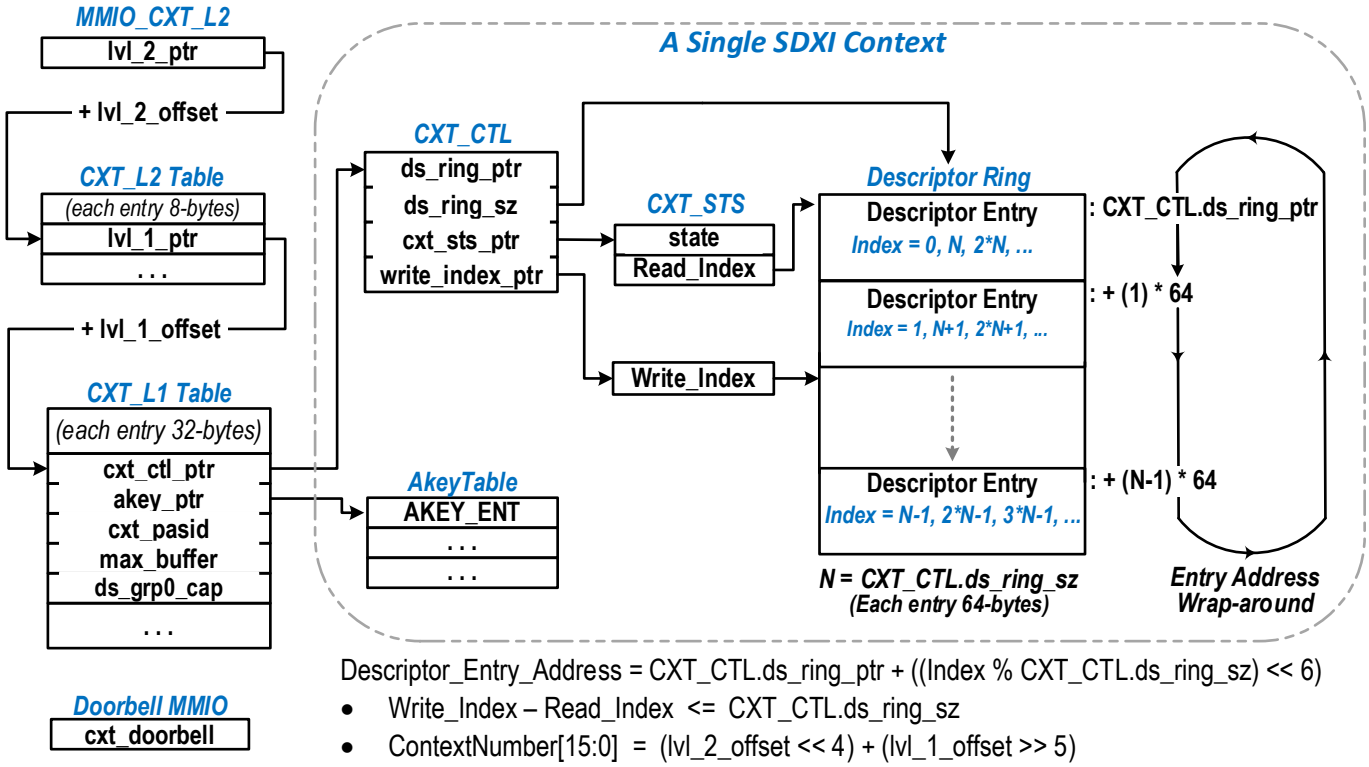
Descriptor "csr" field	Definition
0b	The descriptor requires atomic completion-status mode.
1b	The descriptor requires only simple completion-status mode.

3.

1. Atomic Completion-Status Mode: the producer configures and requires descriptor completion using the atomic completion-status capability.
    a. Privileged software indicates the availability of this mode to a context by setting CXT_CTL.csa to "0"; but shall only do so when the function has both atomic completion-status capability (MMIO_CAP0.cs_cap is "00b" or "10b") and access to interface-supported atomic operations. If the mode is not available, but the context requires the mode, privileged software cannot enable execution of the context.
    b. When the mode is exposed to a context, the producer shall set a descriptor's "csr" ("completion status requirement") field to "0" if the atomic completion-status capability is required for the CST_BLK.
    c. There is no function-enforced terminal value for CST_BLK.signal in this mode; however, it is recommended that the producer use "0" as the terminal value.
    d. When using this mode for a descriptor, the producer may update the CST_BLK atomically even after the associated descriptor has been issued.
    e. If an SDXI function does not support the atomic completion-status capability, the function shall abort and log an error for a descriptor with the "csr" field set to "0".
    f. When atomic CST_BLK access is not required, this mode is not recommended; use the simple completion-status mode instead for greater compatibility across all SDXI implementations.

2. Simple Completion-Status Mode: the producer configures and requires descriptor completion in a manner that complies with the non-atomic completion-status capability, but which may be satisfied by either completion capability.
    a. This mode is always available for a context and is supported by both completion capabilities. Privileged software may recommend that the context use only this mode by setting CXT_CTL.csa to "1". Privileged software must expose this mode when only non-atomic completion-status capability is supported by the SDXI function.
    b. The producer uses this mode by setting a descriptor's "csr" ("completion status requirement") field to "1" to indicate that atomic completion-status of the CST_BLK is not required. Furthermore, the producer shall set the initial value of CST_BLK.signal to "1" before submitting the descriptor. Lastly, the producer detects completion when CST_BLK.signal has the terminal value of "0".
    c. The producer shall not rely upon atomic access of the CST_BLK nor the manner in which the SDXI function modifies it.
    d. The SDXI function shall use the non-atomic completion-status capability for this mode when available; otherwise, it shall use the atomic completion-status capability. This mode is the most compatible mode across all SDXI implementations.
    e. This mode is preferred when both modes are available to the context (CXT_CTL.csa is "0") and a descriptor does not require atomic CST_BLK access.

Figure 5-1: An SDXI Function Context



Descriptor_Entry_Address = CXT_CTL.ds_ring_ptr + ((Index % CXT_CTL.ds_ring_sz) << 6)

- Write_Index – Read_Index <= CXT_CTL.ds_ring_sz
- ContextNumber[15:0] = (lvl_2_offset << 4) + (lvl_1_offset >> 5)

An SDXI function provides programmed-data acceleration by reading and executing a series of memory-based, naturally-aligned, 64-byte "descriptors". Each descriptor's "opcode" field encodes a requested operation and the remainder of the descriptor specifies additional parameters. (Descriptor operations are further discussed in *"5.1, Descriptor Operations"*). Descriptors are placed in a circular ring buffer that starts at a specified address (CXT_CTL.ds_ring_ptr). The ring is contiguous at the translation level configured for the SDXI function. The ring is configured to contain a given number of descriptor entries (CXT_CTL.ds_ring_sz). The number of bytes allocated in memory for the ring is: (CXT_CTL.ds_ring_sz * 64).

The ring and all its related system memory data structures comprise a "context". SDXI uses a 2-level hierarchy of context tables (Context Table Level 2, Context Table Level 1) to enumerate the components of the context. The concatenation of the offsets used to enumerate a context in both Context tables yields the 16-bit "context_number" that is associated with the context. Note that the Context Tables point to a context but are not themselves part of the context. An SDXI function can support multiple concurrent contexts. Contexts are classified into two types: an "unprivileged" type used directly by user applications for data movement; and "administrative" contexts that can be used by privileged software to control all contexts supported by a SDXI function.

An SDXI function may optionally generate DMA requests with PASID when accessing non-context data structures, as well as Context Control Entries and AKey Table Entries. This is controlled through the MMIO_CTL0.fn_pasid_vl and MMIO_CTL0.fn_pasid fields.

A SDXI function may optionally generate DMA requests with PASID when accessing a context's descriptor ring, Write_Index, CXT_STS, and Completion Status Block structures. This is controlled through the CXT_L1_ENT.pv and CXT_L1_ENT.ctx_pasid fields.

A SDXI function may optionally generate DMA requests with PASID when accessing data buffers. This is controlled through the associated AKEY_ENT.pv and AKEY_ENT.pasid fields.

A circular ring requires "start" and "end" indicators. Rather than using memory pointers to track these, a SDXI descriptor ring uses 64-bit *unsigned* logical indices to indicate the start (Read_Index) and end (Write_Index) of the descriptor ring. This simplifies various calculations for SW and HW alike. The logical indices need only be mapped onto descriptor ring addresses when writing or reading a ring entry at a given Index.

- An SDXI descriptor ring shall be contiguous within the address space used to access it. This is determined by the function's PCI requester ID and optional PASID.

CXT_CTL.ds_ring_ptr indicates the start address of the ring within this address space.

The logical Read_Index and Write_Index indices map to ring addresses when writing or reading a ring entry at a given Index. An index N, maps to an address in the ring by:

- ring_entry_address = CXT_CTL.ds_ring_ptr + ((N % CXT_CTL.ds_ring_sz) << 6)

The indices are not expected to reach or exceed (2**64)-1 in practice

After software increments Write_Index and adds entries to the ring, it shall write the context's Doorbell register with the Write_Index value; there is no architectural guarantee that new entries on the ring are processed without this important write.

If software wants to add N entries to the descriptor ring, it must ensure that:

- Write_Index + N - Read_Index <= CXT_CTL.ds_ring_sz

For an enabled context, Read_Index is constantly being updated by the SDXI function as new descriptors are processed. Any value read by software of the Read_Index could be stale, but this is not a problem. Since Read_Index always increases and never wraps in the normal case, a calculation using a stale value of Read_Index will only cause a more conservative understanding of the number of available indices/entries by software. It will never allow enqueing more entries than the ring can hold.

If Write_Index == Read_Index, then the SDXI function does not process more entries of that context until Write_Index and the context's doorbell value are updated beyond Read_Index. The SDXI function will stop processing entries when Read_Index == Write_Index or Read_Index points to an invalid entry. The SDXI function shall never process the entry pointed to by Write_Index or any entry that is logically past Write_Index, regardless of the value of the descriptor's Valid field.

For a valid context, software shall only modify valid descriptors located between the context's Read_Index and Write_Index when the associated CXT_STS.state is CXTV_STOP_[SW, FN] or CXTV_ERR_FN.

5.1 Descriptor Operations

Each descriptor operation has an "opcode" field encoding and a specific format for the bytes that comprise it. The "opcode" field is subdivided into smaller subfields including "type" and "subtype". For the purposes of standardized enumeration and implementation, SDXI arranges sets of related operations into an operation group; each operation within a group has a shared "type" encoding combined with an unique "subtype" encoding. For example, all the operations of the DmaBaseGroup (DSC_DMAB) have a "type" encoding of "0x001".

The enumeration and enabling for most operation groups are provided by a set of operation-group ("op_grp") fields described below.

1. Enumerated support of the operation group by the SDXI function through MMIO_CAP1.opb_000_cap.
2. Availability of the operation group for all SDXI function contexts configured by privileged software through MMIO_CTL2.opb_000_avl.
3. Enabling of the operation group for an individual context configured by privileged software through CXT_L1.opb_000_enb.

Using these fields, SDXI places each descriptor operation group into one of the following categories.

1. Mandatory, not-enumerable: all SDXI functions are required to implement the group which is always available and enabled; there are no assigned operation-group fields. Examples are DmaBaseGrp and AdminGrp.
2. Optional: an SDXI function may implement the operation group; if it does, it shall support the assigned operation-group fields. An example is the AtomicGrp.

When enumeration is defined and true for an operation group at a given bit position "x" in the operation-group fields, an SDXI context may use the operation if the following below condition is true; if not, the SDXI function logs an error and stops execution of the context -- i.e., initiates a StopErr:Cxt action.

- MMIO_CAP1.opb_000_cap[x] & MMIO_CTL2.opb_000_avl[x] & CXT_L1.opb_000_enb[x]

Privileged Software shall not set an operation-group bit in MMIO_CTL2.opb_000_avl and CXT_L1.opb_000_enb if the corresponding bit in MMIO_CAP1.opb_000_cap is "0". The SDXI function may ignore such bits and software shall not rely upon their behavior.

The definition of descriptor opcodes, their formats, and their function operation-group fields are given in *"Chapter 6, SDXI Descriptor and Operation Specification"*.

5.2 Enqueuing one or more Descriptors

The following procedure may be used to enqueue one or more descriptors into a descriptor ring:

1. Check for sufficient space in the descriptor ring by reading the Read_Index, Write_Index and ring size.
2. Reserve space in the descriptor ring by adding a value to the Write Index in memory corresponding to the number of descriptors to be enqueued. In a multi-producer scenario, Write_Index can only be safely updated using an atomic compare-and-swap operation. This operation also provides the required single-threaded guarantees with respect to the update of Write_Index.
3. Let "PWI" refer to the pre-incremented Write_Index. Then Descriptors may be written into memory starting at:

   CXT_CTL.ds_ring_ptr + (PWI % CXT_CTL.ds_ring_sz)*64.

   Software shall ensure that the Valid field in a descriptor header is set to 0 (Invalid) until the whole descriptor is written. The function may start processing the whole descriptor immediately once the valid field is set to 1. The function may read the descriptor one or more times prior to the doorbell being written.
4. Once all descriptors are written to memory, write the updated Write_Index value to the context's Doorbell register.

Figure 5-2: Example x86-64 SDXI Enqueue Code

```
/*  Example code for enqueueing descriptors on an SDXI ring using gcc 9.3 (and later).

    The code should be correct for multiple CPU architectures as it relies solely on: the cpu-independent gcc single-
    threaded guarantees for instruction re-ordering; gcc atomic built-ins which control compiler instruction-reordering
    and proper emitting of CPU memory barriers and instructions; and the behavior of the volatile type keyword to
    ensure that gcc does not optimize away or cache critical SDXI index locations. (Note: the code has been compiled
    at -O3 and the assembly examined on both x86-64 and ARM64 for correctness wrt these issues.)

    Using a fence (__atomic_thread_fence(__ATOMIC_ACQ_REL)) at key points in the code ensures for any given
    thread that: the compiler doesn"t re-order instructions due to optimizations; and the compiler emits the minimum
    but necessary CPU memory barriers to ensure the required read and write ordering within the thread.

    For example, on x86-64 with optimizations enabled, the fence compiles to nothing due to the TSO memory model;
    however, on ARM64 this compiles to "dmb ish". Using __ATOMIC_SEQ_CST instead for the fence penalizes the
    emitted code in some cases -- e.g. mfence on x86-64 which is too strong.

    In a multi-producer scenario, Write_Index can only be safely updated using __atomic_compare_exchange_n().
    This operation also provides the required single-threaded guarantees wrt the update of Write_Index. Using
    _ATOMIC_SEQ_CST here seems like the right generic choice which gcc compiles to the correct CPU instructions
    such as "lock cmpxchg" on x86_64 and "ldaxr, cmp, bne, stlxr" on ARM64.
*/
#define SDXI_DESCR_SIZE      64
#define SDXI_DS_NUM_QW       (SDXI_DESCR_SIZE / sizeof(uint64_t))
#define _ALIGN64             __attribute__ ((aligned (64)))
#define SDXI_MULTI_PRODUCER  1         // Define to 0 if single-producer.
/*  Shows a method for writing an array of descriptors to the SDXI ring given a ring index.

    Method: Skip writing the first QW of the first new entry to be added to the ring, but write all the other entry data as
    normal. This ensures that we don"t trigger the SDXI function into reading incomplete entries as we are writing
    them. Finally, write the first QW of the first new entry to the ring so the SDXI function can start processing the new
    entries.

    This is safe provided that the zeroth bit of the LSB (VALID bit) of the first new ring entry is "0" *before*
    Write_Index is advanced past it (since an SDXI function will not read past an invalid ring entry). This requirement
    is ensured by SW and the SDXI function in the following way: SW shall always initialize at least every 64th byte of
    the ring to zero before first use of the ring; and subsequently, the SDXI function shall always clear the zeroth bit of
    the LSB of every ring entry the function processes.
*/
int update_ring(
   uint64_t * const enq_entries,       // Ptr to entries to enqueue
   uint64_t enq_num,                   // Number of entries to enqueue
   uint64_t * ring_base,               // Ptr to ring location
   uint64_t ring_size,                 // (Ring Size in bytes)/64
   uint64_t  index )                   // Starting ring index to update
{
   uint64_t skip = 1;                  // "skip" writing the first QW.
   for (uint64_t i = 0; i < enq_num; i++){
      uint64_t * ringp = ring_base + ((index + i) % ring_size) * SDXI_DS_NUM_QW;
      uint64_t * entryp = enq_entries + (i * SDXI_DS_NUM_QW);
      for (uint64_t j = skip; j < SDXI_DS_NUM_QW; j++ ){
         *(ringp + j) = *(entryp + j);
      }
      skip = 0;                        // Don"t skip first QW for other entries
   }
   // Now write the first QW of the first new entry to the ring.
   __atomic_thread_fence(__ATOMIC_ACQ_REL);
   *(ring_base + (index % ring_size) * SDXI_DS_NUM_QW) = *enq_entries;
   return 0;
}
```

Figure 5-3: Example x86-64 SDXI Enqueue Code (cont)

```
/*  Enqueues entries onto an SDXI Ring.
    We assume that the OS and/or the application have already setup the SDXI function's context and can pass the
    locations and sizes of the ring, Write_index, Read_Index, and Door_Bell to this routine. Once Write_Index can be
    safely advanced for the desired number of ring entries, this method does so and *then* afterwards populates the
    new ring entries.
*/
int sdxi_enqueue(
   uint64_t * const enq_entries,               // Ptr to entries to enqueue
   uint64_t enq_num,                           // Number of entries to enqueue
   uint64_t * ring_base,                       // Ptr to ring location
   uint64_t ring_size,                         // (Ring Size in bytes)/64
   uint64_t const volatile * const Read_Index, // Ptr to Read_Index location
   uint64_t volatile * const Write_Index,      // Ptr to Write_Index location
   uint64_t volatile * const Door_Bell)        // Ptr to Ring Doorbell location
{
   uint64_t read_idx, old_write_idx, new_idx;
   // SW should do intelligent retry & back-off; while loop shown for simplicity
   while (true){
      // Get Read_Index before Write_Index to always get consistent values
      read_idx  = *Read_Index;
      __atomic_thread_fence(__ATOMIC_ACQ_REL);
      old_write_idx = *Write_Index;
      if (read_idx > old_write_idx){
         // Only happens if Write_Index wraps or ring has bad setup
         return 1;
      }
      new_idx = old_write_idx + enq_num;
      if (new_idx - read_idx > ring_size) {
         continue;                          // Not enough free entries, try again
      }
      if ( SDXI_MULTI_PRODUCER ){
         // Try to atomically update Write_Index before other threads with
         // compare_exchange operation. This built-in also ensures the required
         // single-threaded guarantees wrt the update of Write_Index.
         bool success =
            __atomic_compare_exchange_n( Write_Index, &old_write_idx, new_idx,
                                         false, __ATOMIC_SEQ_CST, __ATOMIC_SEQ_CST );
         if ( success ) break;  // Updated Write_Index, no need to try again.
      }
      else {
         // Single-Producer case
         *Write_Index = new_idx;
         __atomic_thread_fence(__ATOMIC_ACQ_REL);
         break;    // Always successful for single-producer
      }
      // Couldn"t update Write_Index, try again.
   }
   // Write_Index is now advanced. Let's write out entries to the ring.
   update_ring(enq_entries, enq_num, ring_base, ring_size, old_write_idx);

   // Door_Bell write required; only needs ordering wrt update of Write_Index.
   *Door_Bell = new_idx;
   return 0;
}
```

5.2.1 Multi-Producer Enqueue

Multiple producer entities may use the enqueue procedure described in the prior section in parallel without the use of locks or other higher-level serialization methods applied to the descriptor ring. While a context's Write_Index value in memory must be monotonically increasing, the SDXI function may see doorbell writes to a context whose values are not monotonically increasing. The function shall discard such doorbell writes.

5.3 Descriptor Processing

This section describes the specific behavior required of an SDXI function as it processes a descriptor through the phases of parsing, execution, and completion. With respect to each other, descriptors within the same context are parsed in order, but may execute and complete out-of-order or in parallel.

Descriptor Parsing Main Loop:

1. The function derives values for the context's memory-resident CXT_STS.state, Read_Index, Write_Index, and the descriptor at Read_Index.
    a. For each reference to a memory-resident value in this parsing loop, the function may source the value from either an internal function cache or the architected locations in memory as appropriate unless explicitly stated to the contrary.
    b. If during this parsing loop, errors occur when the function reads or writes these values to memory, the function shall log an appropriate error and end this parsing loop by proceeding to step 11.
    c. When the function serializes a value to memory, it shall ensure that any internal source of the value matches its memory-resident value; if not, the function shall update the value in memory.
2. The function shall examine the context's CXT_STS.state and then perform the following actions:
    a. If the context is not at CXTV_RUN, the function shall end this parsing loop by proceeding to step 11.
    b. If the context is at CXTV_RUN, the function may serialize the Read_Index value in memory as appropriate so software can make good forward progress.
3. If the function receives a doorbell_value for the context or fencing was required on the current descriptor and has completed; the function's context parsing shall go to step 4.
    a. The parsing may also go to step 4 for implementation-specific reasons.
    b. Otherwise, the function's descriptor parsing for the context goes back to step 1.
4. The function shall perform checks on the context's Read_Index and Write_Index values as follows.
    a. If Write_Index is less than Read_Index, the function shall log an error and end this parsing loop by proceeding to step 11.
    b. If Write_Index minus Read_Index is greater than the context's descriptor ring size (ds_ring_sz), the function shall log an error and end this parsing loop by proceeding to step 11.
    c. If Read_Index is less than Write_Index, the function's descriptor parsing may proceed to step 5.
    d. Otherwise, the parsing shall go back to step 1.
5. The function shall poll the valid bit of the descriptor at Read_Index until it is valid or until an implementation-specific timeout period has been reached. If timeout happens, the function shall go back to step 1.

6. The function shall check the validity of the descriptor's type, subtype, chain, fence, and sequential fields for the current context. If any are invalid, the function shall log an appropriate error and end this parsing loop by proceeding to step 11. The type and subtype fields specify the descriptor operation that the function shall check for validity by the following steps.
    a. If the function does not support the operation, then the operation is invalid.
    b. If the operation is a supported, AdminGrp descriptor operation, but the current context number is not "0" (Admin Context), then the operation is invalid.
    c. If the operation is a supported operation from a supported optional descriptor group, then there is a corresponding bit field, "x", in the op_grp (operation group) fields. If the following condition is not true, then the operation is invalid:
    d. MMIO_CAP1.opb_000_cap[x] & MMIO_CTL2.opb_000_avl[x] & CXT_L1.opb_000_enb[x]
    e. Otherwise, the operation is valid.
7. The function shall check the fence bit in the descriptor header. If it is set, the function shall poll for previously executing descriptors of the context to complete until an implementation-specific timeout period has been reached.
    a. If timeout happens, the function shall go back to step 1. If the previously executing descriptors complete with error, the function shall end this parsing loop by proceeding to step 11.
    b. If these descriptors complete successfully and CXT_STS.state is CXTV_RUN , the parsing continues to the next step; otherwise parsing retruns to step 1.
8. The function shall ensure that the descriptor's valid field in memory is set to invalid. If the write succeeds, the function goes to step 9.
9. The function shall perform the following in any order or in parallel.
    a. The function shall initiate the background execution of the current descriptor using the execution flow later in this section. The execution may be asynchronous to this parsing loop, thus the function does not need to wait for the execution to complete before parsing new descriptors.
    b. The function shall increment the derived value of Read_Index. Note, Read_Index is not required to be serialized to memory at this time. However, it is recommended that an SDXI function implementation serialize Read_Index frequently enough to allow software to perform simple load balancing between contexts.)
10. The function shall continue descriptor parsing for the context at step 1.

Descriptor Parsing End:

11. If no error has occurred in parsing the descriptor, then descriptor parsing ceases and the function proceeds to the "Execution & Completion Phase".
12. If an error has occurred in parsing the descriptor, the function shall initiate a LogErr:Cxt and StopErr:Cxt action on the context in any order or in parallel. Descriptor parsing ceases and no execution occurs. Note, both the descriptor's valid field and the descriptor's completion status block are not modified in this case.

Execution & Completion Phase:

1. The function shall execute the current descriptor operation including accessing any referenced AKeys, RKeys, and data buffers. DSH may be applied to data buffer accesses; see *"3.6, Data Steering Hints (DSH)"* for more details. The execution is constrained by the following.
    a. If the Sequential Consistency (S) flag is set in the descriptor, all writes from prior descriptor operations in the same context shall be made globally visible prior to making writes from the current descriptor operation globally visible, regardless of address.
2. The function shall consider the execution of the current descriptor finished when all background actions belonging to the descriptor are no longer executing. For each action belonging to the current descriptor, one of the following shall be true.
    a. The action has been fully performed.
    b. The action has encountered an error and was stopped before proceeding further.
    c. The action was stopped or not even begun because of an error in another action belonging to the same descriptor or the entire descriptor was aborted due to the context stopping.
3. When the execution of the current descriptor is finished, the function proceeds to step 4.
4. If no descriptor execution errors occurred, and the descriptor's "np" field is "1", then the descriptor has completed.
5. If no descriptor execution errors occurred, and the descriptor's "np" field is "0", the function shall update the CST_BLK.signal field. If no error occurs in updating the field, then the descriptor has completed. If an error occurs in updating the field, the function starts the SignalErr:Cxt action, which shall set CST_BLK.er and start the StopErr:Cxt and LogErr:Cxt actions in any order or in parallel. The descriptor is considered complete when the StopErr:Cxt action completes. (See *"3.4, Error Log"* for details.)
6. If a descriptor execution error occurred and the descriptor's "np" field is "1", then the function shall start the StopErr:Cxt action and LogErr:Cxt action in any order or in parallel. The descriptor is considered complete when the StopErr:Cxt action completes. (See *"3.4, Error Log"* for details.)
7. If a descriptor execution error occurred and the descriptor's "np" field is "0", then the function shall start the StopErr:Cxt action, LogErr:Cxt action, and DescrErr:Cxt action (which shall set CST_BLK.er and update the CST_BLK,signal fields) in any order or in parallel. The descriptor is considered complete when the StopErr:Cxt action completes. (See *"3.4, Error Log"* for details.)

5.4 Descriptor Ordering and Parallel Execution

An SDXI function may prefetch any number of descriptors from a descriptor ring up to the location prior to that indicated by the Write_Index value in memory. Dynamic modification of valid descriptors located between Read_Index and Write_Index-1 is not supported.

The Fence (F) flag may be used to constrain parallel execution. When the Fence flag is set, the function shall wait for all prior descriptor operations to complete before executing the fenced descriptor operation. When the Fence flag is clear, the descriptor operation may be executed while prior descriptor operations are outstanding.

The Sequential Consistency (S) flag may also affect parallel execution. See section *"5.6, Memory Consistency Model"* for more details.

There are no ordering or consistency requirements for descriptors belonging to different contexts.

5.5 Descriptor Completion

When each descriptor operation has completed, the function will appropriately update the referenced CST_BLK completion signal in memory. Descriptors may complete out of order. It is up to software to assign completion signal locations and initial values as appropriate. Some applications may require precise tracking of descriptors while others may track groups of descriptors.

When a descriptor encounters an error during processing and causes its context to stop, there may be multiple prior and/or subsequent descriptors from the same context being processed in parallel. These may complete independently with or without error.

5.6 Memory Consistency Model

Unless otherwise specified, the order in which the SDXI function reads and writes buffers and other data structures associated with executing an SDXI descriptor is undefined. Software shall not rely upon the ordering of the data transfer within a single descriptor and thus shall not overlap the source buffer, destination buffer, Atomic Return Data, or completion status block.

The SDXI function shall ensure that when clearing the descriptor header Valid bit, it shall be made globally visible ahead of making any writes associated with the descriptor operation globally visible and ahead of making any writes to the descriptor's Completion Status globally visible.

All writes associated with a descriptor operation shall be made globally visible prior to making changes to the descriptor's Completion Status globally visible. Within the Completion Status, changes to Error Recorded (ER) field shall be made globally visible prior to making updates to the descriptor's completion signal globally visible.

If the Sequential Consistency (S) flag is set, all writes from prior descriptor operations in the same context shall be made globally visible prior to making writes from the current descriptor operation globally visible, regardless of address.

If the Sequential Consistency flag is clear, writes associated with the current descriptor operation may be reordered ahead of, and made globally visible prior to, writes associated with prior descriptor operations that have not yet been made globally visible.

Unlike the fence flag, the sequential consistency flag allows multiple descriptors to be executed in parallel as long as the rules around global visibility are maintained.

> Developer Note: This note is regarding the use of Sequential Consistency Flag and PCIe Relaxed Ordering. In general, data buffer writes associated with a single descriptor operation may be issued over the PCIe bus in any order.

If the S flag is clear, all writes associated with the descriptor operation may be issued as Relaxed Ordered (RO=1) writes. Additionally, writes associated with the descriptor operation may be issued ahead of writes associated with prior descriptor operations.

If the S flag is set, all writes associated with a descriptor operation need to be issued as non-relaxed (RO=0) over the PCIe bus, and only after writes associated with all prior descriptor operations have been issued on the bus.

5.7 Descriptor Chaining

The descriptor chaining mechanism may be used for either Extended Descriptors or as part of the Error Log. See *"5.7.1, Extended Descriptors"* and *"3.4, Error Log"* for more details.

A descriptor chain is formed from 2 or more adjacent 64-byte descriptor entries. Each entry in the chain is referred to as a link. Each link is formatted as a standard SDXI descriptor with an SDXI header and footer. Each link except for the last shall set the Chain (C) bit to 1 in their respective headers. The last link in the chain is indicated by a descriptor with the Chain bit clear.

Software shall reserve enough space in the descriptor ring to enqueue the entire chain. The individual descriptor entries forming the chain may be written into the descriptor ring in arbitrary order.

When processing a descriptor chain, an SDXI function shall increment Read_Index by 1 and clear the header Valid bit for each link in the chain.

5.7.1 Extended Descriptors

Extended descriptors may be utilized by operations that require more than the 52 bytes of operand space provided by a single SDXI descriptor. Additional operand space may be created by chaining 2 or more descriptor entries. Each operation definition shall indicate whether the use of an extended descriptor is required.

Other than the Chain bit, extended descriptors utilize the same descriptor header for each link in the chain.

Only the last link in the chain contains a valid descriptor footer with the pointer to the completion status block. The descriptor footers for all other links in the chain are reserved. Only one CST_BLK completion signal update is performed for the entire extended descriptor.

Error log entries are generated for the extended descriptor as a whole.

An SDXI function processing an extended descriptor shall wait until all links are valid before executing the descriptor operation.

5.8 Descriptor Driven Interrupts

Interrupts generated by descriptors do not set any sort of internal MMIO status as may be present in traditional devices. Any interrupt status is expected to be maintained in memory.

For these interrupts, the function does not explicitly know when interrupt service is complete. These interrupts must be edge triggered as the function does not know when to de-assert a level-sensitive interrupt. Additionally, if MSI or MSI-X Pending bits are implemented, the Pending state will not be change from Set to Clear while the associated vector is masked.

6 SDXI Descriptor and Operation Specification

An SDXI function provides programmed-data acceleration by reading and executing a series of memory-based, naturally-aligned, 64-byte "descriptors". Each descriptor's "opcode" field encodes a requested operation and the remainder of the descriptor specifies additional parameters. The format of each supported descriptor is defined in this chapter. The operation of the descriptor ring and the processing, execution, and completion of descriptors is discussed in *"Chapter 5, SDXI Descriptor Ring Operation"*.

For the purposes of standardized enumeration and implementation, SDXI arranges sets of related descriptor operations into an operation group; each operation within a group has a shared "type" encoding combined with an unique "subtype" encoding. The enumeration and enabling for most operation groups are provided by a set of operation-group ("op_grp") fields implemented in MMIO_CAP1.opb_000_cap, MMIO_CTL2.opb_000_avl, CXT_L1.opb_000_enb. The mechanism by which these fields are used is described in *"5.1, Descriptor Operations"*; their encoding for specific operation groups are defined in *"Table 6–1: SDXI Operation* Groups*"*.

Table 6–1: SDXI Operation Groups

Operation-Group Fields

Bits	Mapping	
0	Reserved	
1	Reserved	
2	Reserved	
3	Maps to	⇨
4	Maps to	⇨
5	Maps to	⇨
27-6	Maps to	⇨
31:28	Reserved	

Operation Group Definitions

Operation Group	Type	Required for SDXI Function?
(Reserved)	0x000	Reserved
DMA Base Operation Group (DmaBaseGrp)	0x001	Mandatory, not-enumerable
Administrative Operation Group (AdminGrp)	0x002	Mandatory, not-enumerable; supported in context 0 only.
Full Atomic Operation Group (AtomicGrp)	0x003	Optional, Requires Enablement
Interrupt Operation Group (IntrGrp)	0x004	Optional, Requires Enablement
Minimal Atomic Operation Group	0x003 (not 0x005)	Optional, Requires Enablement
Reserved	0x01B - 0x006	Reserved
Vendor-Defined	0x01F - 0x01C	Optional, Requires Enablement
Reserved	0x7F6 – 0x020	Reserved
Reserved for Error Log	0x7F7	Mandatory, not-enumerable
Reserved	0x7F8 – 0x7FF	Reserved

Table 6–2: SDXI Operation Groups, Types, and Subtypes

Operation Group	Type	Subtype	Operation
(Reserved)	0x000		
DMA Base Operation Group (DmaBaseGrp)	0x001	0x01	DMAB_NOP
		0x02	DMAB_WRT_IMM
		0x03	DMAB_COPY
		0x04	DMAB_REPCOPY
Administrative Operation Group (AdminGrp)	0x002	0x00	DSC_FN_UPD
		0x01	DSC_CXT_UPD
		0x02	DSC_AKEY_UPD
		0x03	DSC_CXT_START_NM
		0x04	DSC_CXT_STOP
		0x05	DSC_ADM_INTR
		0x06	DSC_SYNC
		0x07	DSC_RKEY_UPD
		0x08	DSC_CXT_START_RS
Full Atomic Operation Group (AtomicGrp)	0x003	0x00	Reserved
		0x01	DSC_ATM_SWAP[1]
		0x02	DSC_ATM_UADD[1]
		0x03	DSC_ATM_USUB
		0x04	Reserved
		0x05	DSC_ATM_AND
		0x06	DSC_ATM_OR
		0x07	DSC_ATM_XOR
		0x08	DSC_ATM_SMIN
		0x09	DSC_ATM_SMAX
		0x0A	DSC_ATM_UMIN
		0x0B	DSC_ATM_UMAX
		0x0C	DSC_ATM_UCLAMPI
		0x0D	DSC_ATM_UCLAMPD
		0x0E	DSC_ATM_CMPSWAP[1]
Interrupt Operation Group (IntrGrp)	0x004	0x00	DSC_INTR
Reserved	0x005		Reserved
Reserved	0x01B - 0x006		Reserved
Vendor-Defined Operation	0x01F - 0x01C		
Reserved	0x7F6 – 0x020		
Reserved for Error Log	0x7F7		
Reserved	0x7F8 – 0x7FF		Reserved

1. *Also supported when Minimal Atomic Operations are present.*

6.1 Descriptor Format for SDXI Operations

6.1.1 Common Header and Footer

All SDXI operations use the following common header and footer encodings in their descriptor format.

Figure 6-1: Common Header and Footer

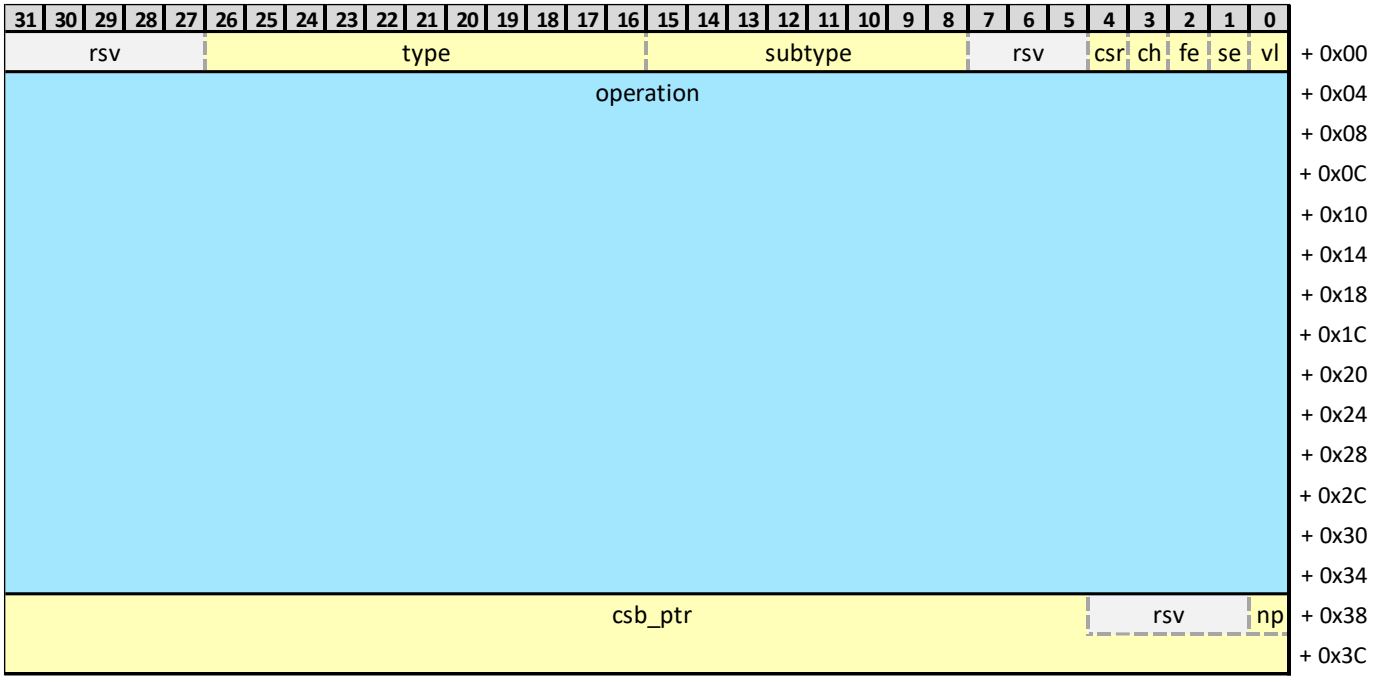


Table 6–3: DSC_GENERIC[^3] SDXI Descriptor Common Header and Footer Format

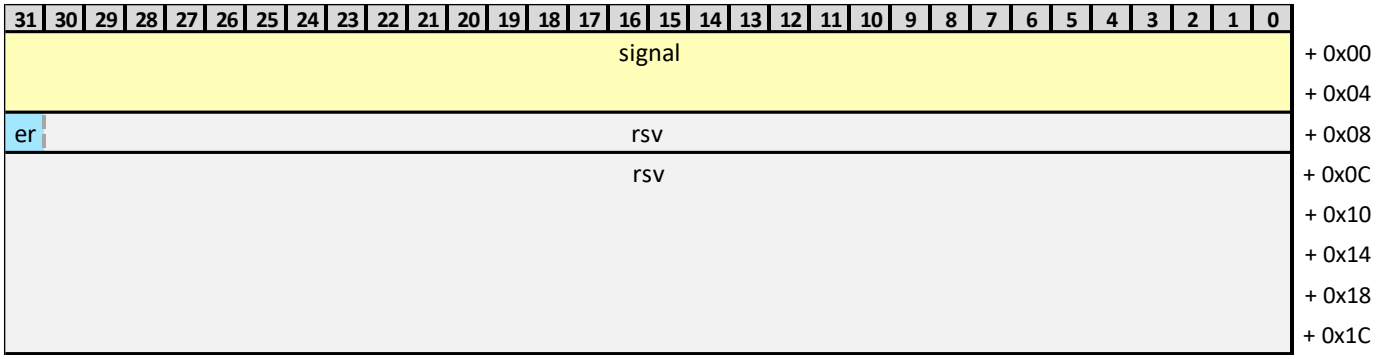
Field	Bits	Subfield	Description
u32 opcode;	000	vl	Valid. 1 = Descriptor is valid. All fields may be processed. 0 = Descriptor is invalid. All other fields within the descriptor shall be ignored. The function shall poll the descriptor until it becomes valid or until an implementation defined timeout period has expired at which point an error is logged.
	001	se	Sequential Consistency 1 = Operation writes are sequentially consistent 0 = Operation writes are not required to be sequentially consistent See "5.6, Memory Consistency Model" for more details.
	002	fe	Fence. See *"5.4, Descriptor Ordering and Parallel Execution"* for more details. 1 = All prior descriptor operations shall complete before executing this descriptor's operation 0 = Execution of this descriptor's operation is permitted prior to the completion of prior descriptor operations
	003	ch	Chain. See *"5.7, Descriptor Chaining"* for more details. 1 = Start or middle of a set of chained descriptors. 0 = End of chain, or non-chained descriptor.
	004	csr	Completion Status Mode Requirement for the descriptor. See *"4.4.2, Completion-Status Modes"* for details. 0 = Atomic completion-status mode shall be used. 1 = Simple completion-status mode shall be used.
	007:005	rsvd	Shall be set to 0
	015:008	subtype	Specifies an operation within the Operation Group.
	026:016	type	Specifies the Operation Group.
	031:027	rsvd	Shall be set to 0
u8 operation[52];	447:032		This region is defined separately for each operation.
u64 csb_ptr;	448	np	no_pointer. When 0, the SDXI function shall update the completion status blocked pointed to by csb_ptr. When 1, there is no completion status block associated with the descriptor to update and the csb_ptr shall be ignored. This mechanism avoids the overhead of completion-block-status processing for a series of descriptor requests that are ordered before a terminal one which will specify a completion status block.
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. A pointer to a 32B aligned region of memory containing the Completion Status block. An invalid pointer may result in an error.

6.1.2 Completion Status Block

When a descriptor's "np" ("no_pointer") field is "0", the descriptor's "csb_ptr" field points to a completion status block ("CST_BLK") whose fields are used by the SDXI function to update (modify) the descriptor operation's completion status and report relevant errors. (The manner in which the SDXI function updates the CST_BLK during descriptor processing is discussed in *"5.3, Descriptor Processing"*.) The producer of the descriptor initializes the CST_BLK before submitting the descriptor.

As discussed in *"4.4.2, Completion-Status Modes"*, the producer may require or be required to use one of two producer completion modification modes for the context with respect to the atomic modification of the CST_BLK. The availability of the modes is reported by privileged software through the CXT_CTL.csa ("completion-status mode availability") field and may be reported to the context by means outside the scope of the SDXI specification. The producer complies to and specifies the mode to the SDXI function by setting the descriptor's "csr" ("completion status requirement") field.

Figure 6-2: Completion Status Block



The completion signal field within the CST_BLK is used by the producer to track the completion of one or more descriptors. It is updated by the SDXI function as appropriate for the completion-status mode indicated by the descriptor's "csr" field. When the signal reaches an appropriate terminal value, the producer may conclude that the associated descriptors have finished and may rely upon the other CST_BLK fields to determine success, failure, or other associated information.

Table 6–4: CST_BLK[^3] (Completion Status Block)

Field	Bits	Subfield	Description
u64 signal;	063:000		Completion Signal value.
u32 flags;	094:064	rsvd	Shall be set to 0
	095	er	Error Recorded (er). By initializing this field to "0" before issuing descriptors associated with the CST_BLK, software can rely upon this behavior: when "1", at least one completed descriptor has encountered an error; when "0", all completed descriptors have not encountered an error. During the processing of an associated descriptor, an SDXI function shall operate on this field as follows: if the descriptor succeeds, the value of the field shall be preserved; and if the descriptor encounters an error, the value of the field shall be "1".
u8 rsvd_0[20];	255:096		Shall be set to 0

Multiple descriptors may point to the same single CST_BLK when atomic completion-status mode is available and used for all associated descriptors. The producer shall refrain from this usage when the

mode is not available. During this usage, if any descriptor associated with the CST_BLK does not specify atomic completion-status mode, the SDXI function may return indeterminate contents of the CST_BLK; the producer shall avoid and not rely upon this behavior.

For each descriptor error reported through CST_BLK, the SDXI function shall ensure that the "er" ("error recorded") field is set to "1" before modifying the completion signal value. When multiple descriptors are associated with a single CST_BLK, the SDXI function may set the "er" field to "1" for errors encountered on one or more of the descriptors.

6.1.3 Attribute Field

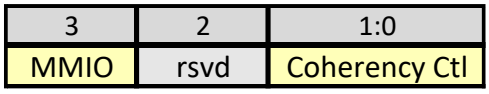
The attribute (Attr) field controls features related to accessing a data buffer. Within a descriptor definition, a postfix may be added to the field name to uniquely identify a specific data buffer (ex. Attr (Src), Attr (Dst)).

Subsequently defined descriptors may contain data buffer pointers which have a corresponding Attr field.

Table 6–5: Attribute Field

Bits	Field	Description
1:0	CoherencyControl	00b : The associated memory location is accessed as I/O coherent. 01b : The associated memory location is accessed as non-coherent. 10b : The associated memory location is accessed as I/O coherent. DSH information is requested to be included when accessing this memory location. See *"3.6, Data Steering Hints (DSH)"* for more details. All other encodings reserved
2	rsvd	Shall be set to 0
3	MMIO	1b: The referenced memory buffer may be located in either system memory or MMIO space. 0b: The reference memory buffer is located in system memory and is not located in an MMIO space. This setting may optimize performance in some implementations.

Figure 6-3: AttributeFormat



6.2 DMA Base Operations Group (DmaBaseGrp)

All SDXI implementations are required to support the DmaBaseGrp operations.

6.2.1 DmaBaseGrp: DSC_DMAB_NOP

The DmaBaseGrp's Nop operation performs no functional DMA operation. A descriptor specifying the operation is processed and ordered the same as other descriptors, obeys all relevant bits in the header, and generates the completion signal.

Figure 6-4: DSC_DMAB_NOP Descriptor Format

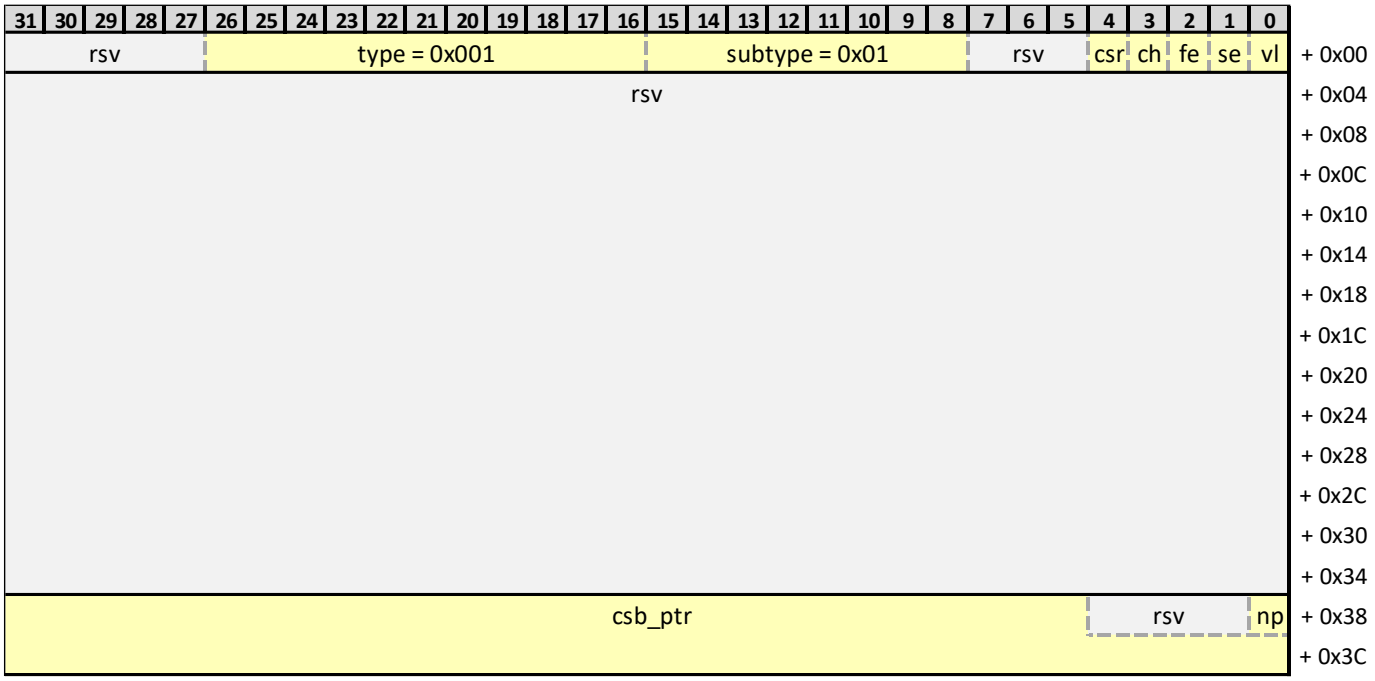


Table 6–6: DSC_DMAB_NOP[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = 0	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x01	See section *"6.1.1".*
	026:016	type=0x001	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 rsvd_0[52];	447:032		Shall be set to 0.
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.2.2 DmaBaseGrp: DSC_DMAB_WRT_IMM Operation

The DmaBaseGrp's WriteImm operation is used to write up to 32 contiguous bytes starting at a destination memory address. The data is supplied inside the descriptor. Following little-endian conventions, data byte 0 is written to the first byte of the destination address. Write atomicity guarantees are implementation specific.

Figure 6-5: DSC_DMAB_WRT_IMM Descriptor Format

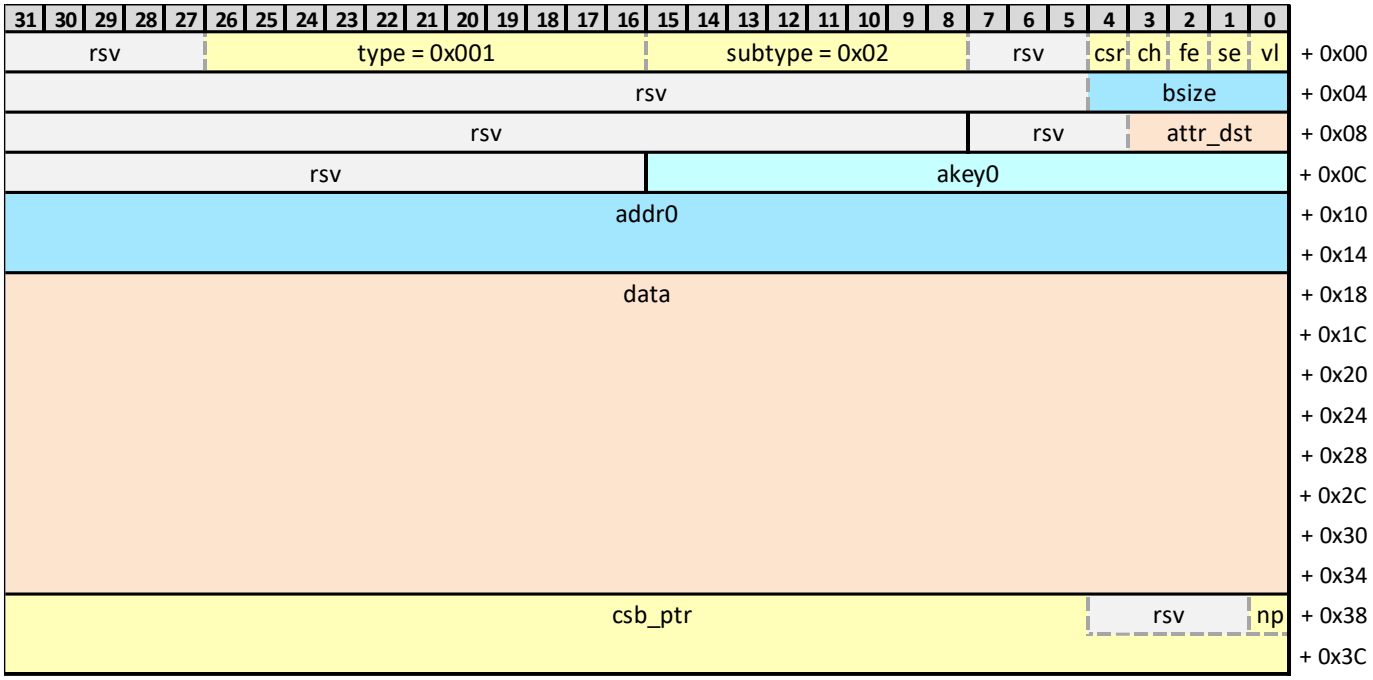


Table 6–7: DSC_DMAB_WRT_IMM[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".*
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x02	See section *"6.1.1".*
	026:016	type=0x001	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u32 size;	036:032	bsize	Specifies the number of bytes to write minus 1. 0x0 = 1 Byte, . . .0x1F = 32 Bytes
	063:037	rsvd	Shall be set to 0
u8 attr;	067:064	attr_dst	Destination buffer attributes
	071:068	rsvd	Shall be set to 0
u8 rsvd_0[3];	095:072		Shall be set to 0.
u16 akey0;	111:096	akey0_dst	Destination buffer AKey
u8 rsvd_1[2];	127:112		Shall be set to 0.
u64 addr0;	191:128	(dst)	Destination buffer starting address
u8 data[32];	447:192		32-bytes of immediate data.
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.2.3 DmaBaseGrp: DSC_DMAB_COPY Operation

The DmaBaseGrp's Copy operation copies a source data buffer to a destination data buffer.

It is an error (Generic Descriptor Error/Unsupported Field Encoding) for the source and destination buffers to exceed the size described by CXT_L1_ENT.max_buffer.

Figure 6-6: DSC_DMAB_COPY Descriptor Format

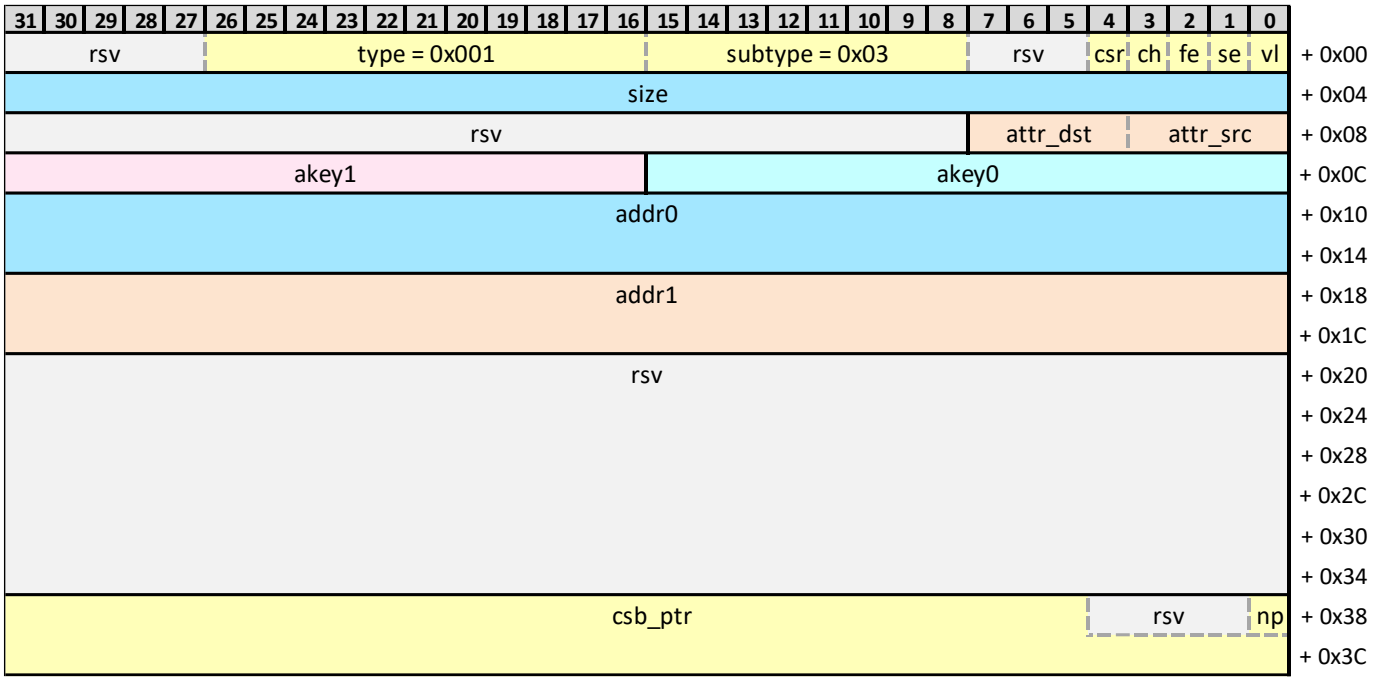


Table 6–8: DSC_DMAB_COPY[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".*
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x03	See section *"6.1.1".*
	026:016	type=0x001	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u32 size;	063:032		Specifies the number of bytes to write minus 1. 0x0 = 1 Byte, . . ., 0xFFFF_FFFF = 2**32 Bytes
u8 attr;	067:064	attr_src	Source buffer attributes
	071:068	attr_dst	Destination buffer attributes
u8 rsvd_0[3];	095:072		Shall be set to 0.
u16 akey0;	111:096	(src)	Source buffer AKey
u16 akey1;	127:112	(dst)	Destination buffer AKey
u64 addr0;	191:128	(src)	Source buffer starting address.
u64 addr1;	255:192	(dst)	Destination buffer starting address
u8 rsvd_1[24];	447:256		Shall be set to 0.
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.2.4 DmaBaseGrp: DSC_DMAB_REPCOPY Operation

The DmaBaseGrp's Repeat Copy operation copies a single 4-KiB-aligned source data buffer to multiple adjacent locations within a 4-KiB-aligned destination buffer. This operation may be used, for example, to fill a large region of memory.

It is an error (Generic Descriptor Error/Unsupported Field Encoding) to specify a total destination size (4Kbytes * (nsize+1) * (num+1)) greater than the size described by CXT_L1_ENT.max_buffer.

Figure 6-7: DSC_DMAB_REPCOPY Descriptor Format

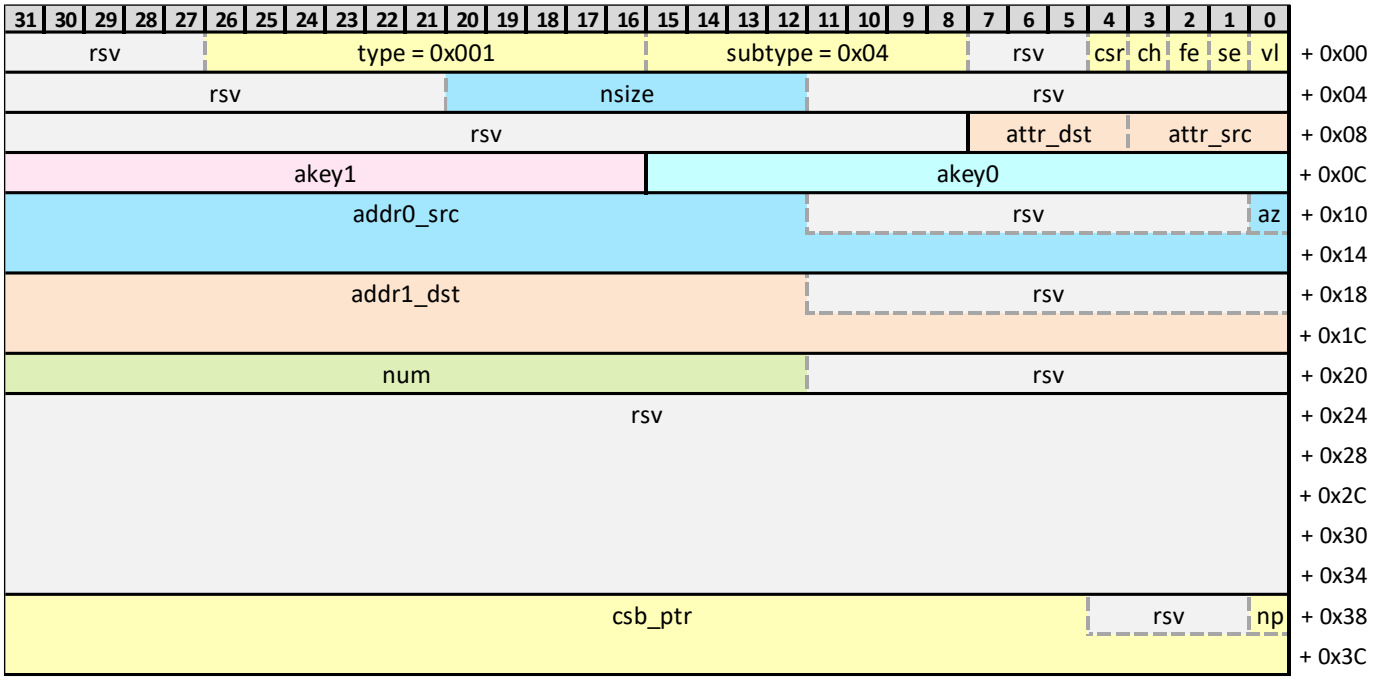


Table 6–9: DSC_DMAB_REPCOPY[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".*
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x04	See section *"6.1.1".*
	026:016	type=0x001	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u32 size;	043:032	rsvd	Shall be set to 0
	052:044	nsize	The size of the source data buffer in 4-Kbyte increments minus 1. 0x0 = 4 Kbytes, … , 0x1FF = 2 Mbytes
	063:053	rsvd	Shall be set to 0
u8 attr;	067:064	attr_src	Source buffer attributes
	071:068	attr_dst	Destination buffer attributes
u8 rsvd_0[3];	095:072		Shall be set to 0.
u16 akey0;	111:096	(src)	Source buffer AKey
u16 akey1;	127:112	(dst)	Destination buffer AKey
u64 addr0;	128	az	"All Zero" When az=1, this indicates that the source buffer is all zero. This allows the SDXI function to optimize the transfer (even avoiding reading the buffer). When az=0, no such guarantee is made. When az=1, SW shall always initialize the entire source buffer to zero; otherwise, the contents of the destination are not defined – and no error is required to be reported in this case.
	139:129	rsvd	Shall be set to 0.
	191:140	addr0_src	4KB aligned source buffer starting address.
u64 addr1;	203:192	rsvd	Shall be set to 0.
	255:204	addr1_dst	4KB aligned destination buffer starting address.
u32 repeat;	267:256	rsvd	Shall be set to 0.
	287:268	num	Specifies the number of times the source buffer is copied to the destination buffer. 0x0 = 1 copy …. 0xF_FFFF = 2**20 copies. The total destination size is: 4Kbytes * (nize+1) * (num+1).
u8 rsvd_1[20];	447:288		Shall be set to 0.
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.3 Atomic Operation Group (AtomicGrp)

This operation group performs various atomic operations. All AtomicGrp operations use the same descriptor format. An SDXI implementation may support these atomic operations or not. If supported, an SDXI Implementation may support the "Minimal" set or the "Full" set -- which includes the minimal set. If the SDXI function supports the full set, it shall not report support for the minimal set. Both sets use the same descriptor "type" and "subtype" for the minimal set.

Privileged software shall verify ***both*** of the following before exposing a set of atomic operations to an SDXI context. (See as discussed in *"4.4, Atomic Operation Support"*)

1. Interface-supported atomic operations. The relevant platform communication interface to the SDXI function is capable and enabled to support atomic accesses to and from the SDXI function.
2. Function-supported atomic operations. The SDXI implementation supports the relevant set of atomic operations.

Figure 6-8: DSC_ATM Descriptor Format

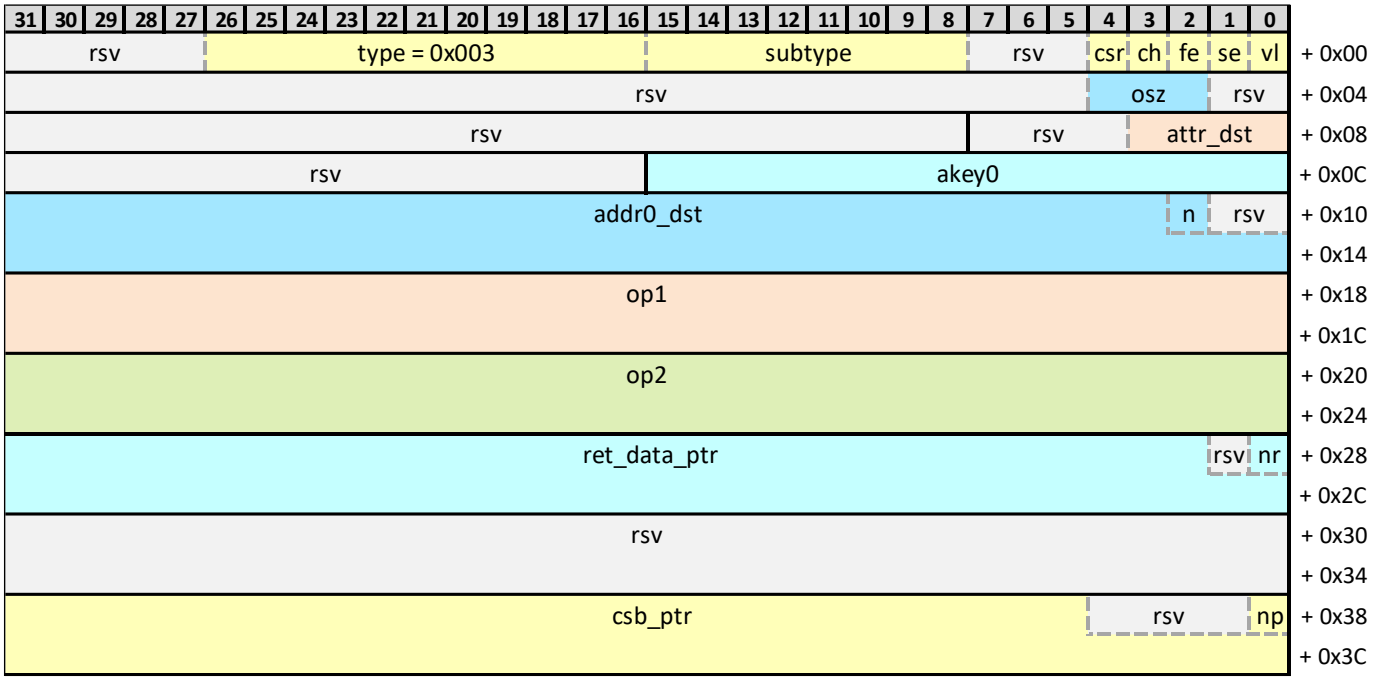


Table 6–10: DSC_ATM[^7] Operation Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".*
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype = SWAP:0x1, UADD:0x2, USUB:0x3, AND:0x5, OR:0x6, XOR:0x7, SMIN:0x8, SMAX:0x9, UMIN:0xA, UMAX: 0xB, UCLAMPI: 0xC, UCLAMPD:0xD, CMPSWAP:0xE	
	026:016	type=0x003	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u32 size;	033:032	rsvd	Shall be set to 0
	036:034	osz	The size of the operands in 4-byte increments minus 1: 000b = 4 bytes, 001b = 8 bytes. All other encodings reserved
	063:037	rsvd	Shall be set to 0
u8 attr;	067:064	attr_dst	Destination buffer attributes
	071:068	rsvd	Shall be set to 0.
u8 rsvd_0[3];	095:072		Shall be set to 0.
u16 akey0;	111:096	(dst)	Destination buffer AKey
u8 rsvd_1[2];	127:112		Shall be set to 0.
u64 addr0;	129:128	rsvd	Shall be set to "00b" since addr0 must always be either 4-byte or 8-byte aligned.
	130	n	Shall be set to "0b" for 8-byte alignment if address0 is an 8-byte operand (osz = "001b")..
	191:131	addr0_dst	Destination buffer memory address to target for the atomic operation. The address shall be aligned to the operand size.
u64 op1;	255:192		Operand 1 Data. If the operand size is set to 4 bytes, the lower 32 bits of Operand1Data are utilized.
u64 op2;	319:256		Operand 2 Data. If the operand size is set to 4 bytes, the lower 32 bits of op2 are utilized.
u64 ret_data_ptr;	320	nr	No Return (nr) When 0, the SDXI function shall return the original memory value at address0 to the location pointed to by ret_data_ptr. When 1, ret_data_ptr shall be ignored, and the original memory value at address0 will not be returned. Software will set ret_data_ptr to 0 when setting nr to 1.
	321	rsvd	Shall be set to 0.
	383:322	ret_data_ptr	Pointer to Atomic Return Data, a naturally aligned operand size location in memory. The original memory value at address0 will be written back to this location when the nr bit is clear. If the nr bit is set, no data is returned.
u8 rsvd_2[8];	447:384		Shall be set to 0.
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

Table 6–11: Atomic Operations Sub-Type[1,2,3,4]

Sub Type	Operation Subset	Operation
0x01	Full, Min	Swap (DSC_ATM_SWAP): *address0 = operand1;
0x02	Full, Min	Unsigned Add (DSC_ATM_UADD)[7]: *address0 += operand1;
0x03	Full	Unsigned Subtract (DSC_ATM_USUB): *address0 -= operand1;
0x05	Full	AND (DSC_ATM_AND): *address0 &= operand1;
0x06	Full	OR (DSC_ATM_OR): *address0 |= operand1;
0x07	Full	XOR (DSC_ATM_XOR): *address0 ^= operand1;
0x08	Full	Signed Min (DSC_ATM_SMIN)[5]: if (*address0 > operand1) { *address0 = operand1; }
0x09	Full	Signed Max (DSC_ATM_SMAX)[5]: if (*address0 < operand1) { *address0 = operand1; }
0x0A	Full	Unsigned Min (DSC_ATM_UMIN)[5]: if (*address0 > operand1) { *address0 = operand1; }
0x0B	Full	Unsigned Max (DSC_ATM_UMAX)[5]: if (*address0 < operand1) { *address0 = operand1; }
0x0C	Full	Unsigned Increment (DSC_ATM_UINC): *address0 = (*address0 >= operand1) ? 0 : (*address) + 1;
0x0D	Full	Unsigned Decrement (DSC_ATM_UDEC): *address0 = (*address0 == 0 || *address0 > operand1) ? operand1 : (*address - 1);
0x0E	Full, Min	Compare and Swap (DSC_ATM_CMPSWAP)[5]: if (*address0 == operand1) { *address0 = operand2; }
All other encodings are Reserved		

1. *The sequence of reading and writing (or releasing) address0 by the SDXI function must be atomic with respect to operations of other agents on address0.*
2. *"*nnn" means the memory contents pointed to by nnn.*
3. *All operations are performed at the specified operand size; all operands must be naturally aligned to that size.*
4. *When the "nr" field is zero, the SDXI function shall write the previous destination value (original memory value at address0) to the location pointed to by the ret_data_ptr field.*
   *if (nr == 0){ *ret_data_ptr = *address0; }*
5. *Note: Provided that the atomicity requirements are met, an SDXI implementation is permitted to write back the unchanged content of address0 when the condition is false.*
6. *The SDXI function may perform the write to the Atomic Return Data location (pointed to by the "ret_data_ptr" field) in any order with respect to the atomic update of the destination location. The SDXI function shall ensure that both the Atomic Return Data write and the destination update are completed and made globally visible before updating the descriptor's completion status block.*
7. *Note that DSC_ATM_UADD can effectively perform a subtract by supplying the two's complement of the value to subtract as the argument value to the operation since overflow is not detected.*

6.4 IntrGrp Operation Group

6.4.1 IntrGrp DSC_INTR Operation

The IntrGrp's Interrupt operation is used to generate an interrupt. The interrupt number and address space are specified in the referenced AKey.

Figure 6-9: DSC_INTR Descriptor Format

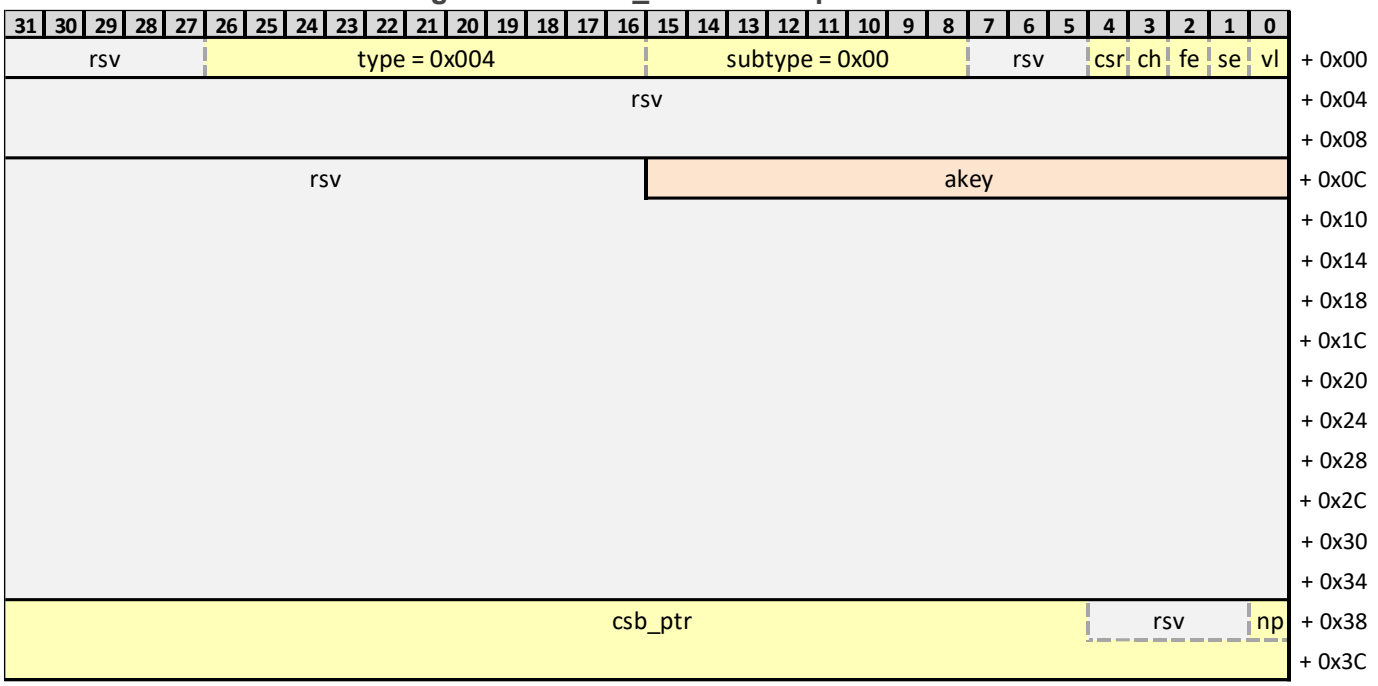


Table 6–12: DSC_INTR[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1"*.
	001	se = *	Sequential Consistency. See section *"6.1.1"*.
	002	fe = *	Fence. See section *"6.1.1"*.
	003	ch = 0	Chain. See section *"6.1.1"*. Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x00	See section *"6.1.1"*.
	026:016	type=0x004	See section *"6.1.1"*.
	031:027	rsvd	Shall be set to 0
u8 rsvd_0[8];	095:032		Shall be set to 0.
u16 akey;	111:096		AKey used for this operation
u8 rsvd_1[42];	447:112		Shall be set to 0.
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1"*.
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1"*.

6.5 Vendor Defined Operation Group

The desired expansion model for new SDXI function operations is to create a new architectural operation group within this specification with architected operations. This prevents fragmentation of the architecture. But there may be rare circumstances where an SDXI function vendor has a need to create a non-standard, vendor-specific operation. Such an operation shall use the format of the DSC_VENDOR descriptor. Using a standard format allows leveraging of the standard software ecosystem for SDXI.

Figure 6-10: DSC_VENDOR Descriptor Format

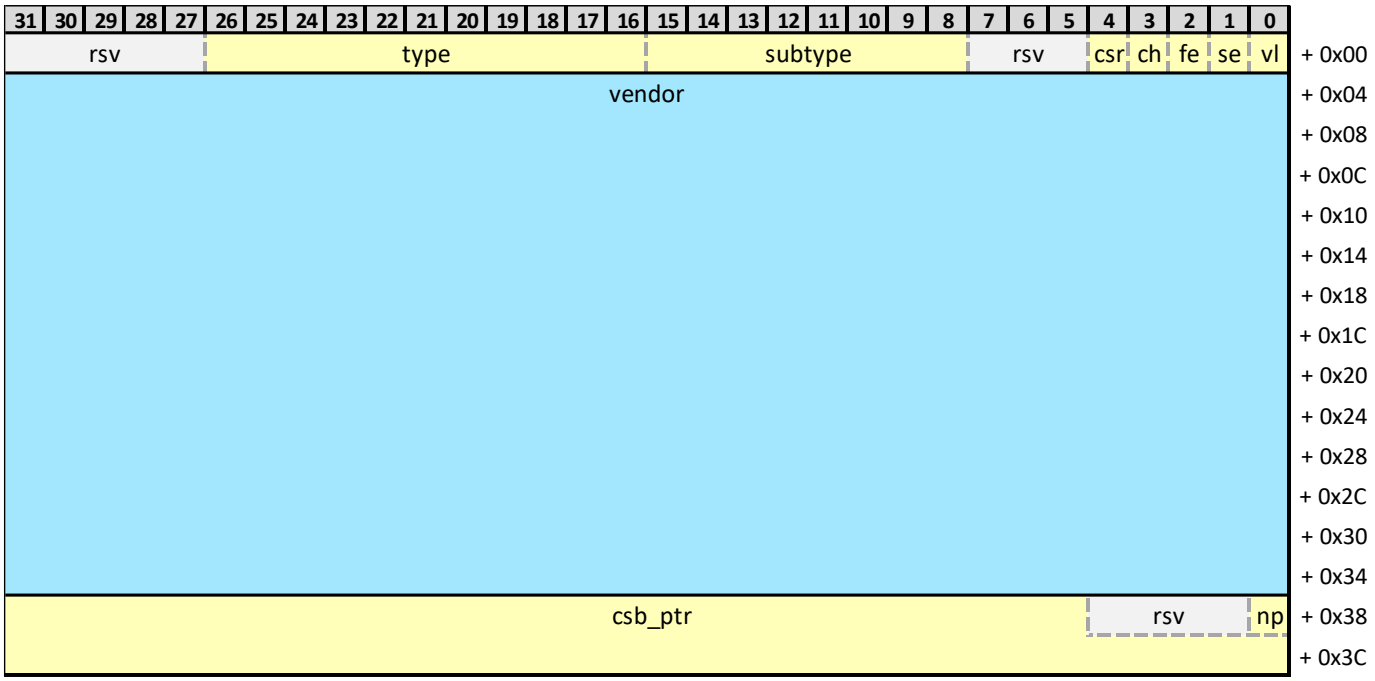


Table 6–13: DSC_VENDOR[^3] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".*
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype	Vendor-Defined.
	026:016	type	Vendor-Defined but must be between 0x01F – 0x01C.
	031:027	rsvd	Shall be set to 0
u8 vendor[52];	447:032		These bits are defined by the vendor based on the VendorID and SubType fields
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6 Administrative Operation Group (AdminGrp)

Administrative operations are used by privileged software to manage the SDXI function. A key set of such operations comprise the Administrative Operation Group (AdminGrp). Software shall issue AdminGrp operations only in Context 0 ("Administrative" Context) within each function; the SDXI function shall generate a context error if an AdminGrp operation is issued on any other context. AdminGrp operations are described below and in the following sections.

- The AdminGrp's DSC_UPD_[FN, CXT, AKEY, RKEY] update operations are used by privileged software to indicate changes to various memory data structures. Update operations shall be used even when transitioning data structures from the invalid to valid state. This is done to simplify software-emulated implementations.
- The AdminGrp's DSC_SYNC operation is used as a barrier after one or more update operations. After completion of the DSC_SYNC, the effects of prior AdminGrp updates to the structures selected by the DSC_SYNC operation shall be seen by all new descriptors.
- The AdminGrp's DSC_CXT_START_[NM, RS] and DSC_CXT_STOP operations may be used to control the execution state of a target context.
- The AdminGrp's DSC_ADM_INTR operation is used to generate interrupts from the local function hosting the administrative context.

6.6.1 Accessing Contexts, Akey Tables, and RKey Table by Index

An Administrative operation may operate on a context entry, an entry in an Akey table, or an entry in the RKey table; each of these entries are referenced by an index within their containing structure. For the function to operate on an indexed entry, the index must be within the specific limit for that structure. The function shall log an error when an operation specifies an index that is outside of the appropriate limits.

Many administrative operations may be applied to a range of entries within a data structure. The range is specified explicitly in the descriptor as a starting index "_start" field and an ending index "_end" field; implicitly the relevant index limit for the structure is also used. (For example, to operate on contexts 1 through 127 inclusive, software would set cxt_start to "1" and cxt_end to "127".) The range of the specified entries is operated upon as shown in the following pseudo-code.

Figure 6-11: Operation on Range of Entries

```
// Let entry_type indicate the type of referenced data-structure (CXT, AKEY, RKEY)

   if ( _start > _end
         || ( entry_type == CXT
               && ( MMIO_CTL2.max_cxt  >  MMIO_CAP1.max_cxt
                     || _end  >  MMIO_CTL2.max_cxt
                  )
            )
         || ( entry_type == AKEY
               && ( MMIO_CTL2.max_akey_sz  >  MMIO_CAP1.max_akey_sz
                     || CXT_L1_ENT.akey_sz  >   MMIO_CTL2.max_akey_sz
                     || _end  >=  2**(8 + CXT_L1_ENT.akey_sz)
                  )
            )
         || ( entry_type == RKEY
               && ( MMIO_RKEY.tbl_sz  >  MMIO_CAP0.max_rkey_sz
                     || _end  >=  2**(8 + MMIO_RKEY.tbl_sz)
                  )
            )
   ) {
         error();
   }

   // Asynchronous, Parallel Issue of Operation on Entries
   index = _start;
   while ( index <= _end ) {
      async_issue_operation( index );
      index++;
   }
```

6.6.2 Targeting Multiple Contexts with A Single Administrative Operation

Many administrative operations are defined such that a single operation may target the entire function or selected multiple contexts. For such an operation, the SDXI function may evaluate individual target contexts in any order and in parallel. The following describes the resulting architectural behavior.

Evaluating Contexts:

1. A successful completion or error result from operating on one context is unordered with respect to another targeted context.
2. The SDXI function may terminate an operation due to an individual context error at any point and may skip evaluation of remaining contexts. It is implementation-specific if an error detected in one context affects whether any other targeted context is evaluated.

Returning Results and Completion Status:

1. If the administrative operation results in an architectural state change or explicit synchronization barrier (DSC_CXT_START_[NM, RS], DSC_CXT_STOP, and DSC_SYNC), then the completion status is signaled in the following way. (Note, each operation shall specify which background actions-must be completed for the function to signal completion.)
    a. If all target contexts have been operated upon successfully without error, the operation returns a successful completion status block.
    b. If the operation encounters an error in at least one context, the operation will return an unsuccessful completion. Note, an operation shall only log one error even if the operation evaluates multiple contexts with error.
    c. Regardless of overall success or failure, the SDXI function shall cease evaluating targeted contexts before generating a Completion Status Block and, if required, logging an error.
2. If the administrative operation is a data-structure update operation to any non-coherently cached copies of function state (DSC_ADM_UPD_[FN, CXT, AKEY, RKEY]), then the completion status merely signals that the function has begun the update action. Completion of the operation does not imply completion of the requested update action(s). Further behavior follows for the update action(s).
    a. Performing an update action on an invalid context shall have no effect and logs no error.
    b. If all target contexts have been acted upon successfully without error, then no error log is generated.
    c. If any update action encounters an error in at least one context, the update action will log an error. It is implementation-dependent if an update action may encounter an error and if multiple update action errors log one or more errors.

A number of operations are idempotent -- viz. they may be executed several times with the same input without changing the final result beyond its first execution. This may be useful if an operation targeting multiple contexts has failed and needs to be resubmitted after remediation. Consult the description of the actual operation for more details.

6.6.3 AdminGrp DSC_CXT_START_[NM, RS] Operations

The AdminGrp's DSC_CXT_START_[NM, RS] operation initiates background actions that start all valid target contexts by changing each context's CXT_STS.state value from CXTV_STOP_[SW, FN] to CXTV_RUN. The targeted contexts are described using the cxt_start and cxt_end fields. The operation may also optionally signal the context to run using the supplied doorbell_value. The doorbell_value is evaluated per the rules specified in *"4.3.3, Doorbell Register and Context Signaling"*.

DSC_CXT_START_NM is intended to be used by the privileged software that administers the target context. The operation transitions CXTV_STOP_SW, CXTV_STOP_FN, and CXTV_RUN to CXTV_RUN for valid contexts; it will log an error if the target context is not in one of the above states for valid contexts.

DSC_CXT_START_RS is intended to be used by hypervisor and privileged software to restore a context to CXTV_RUN that had been previously suspended to CXTV_STOP_FN when the function transitioned out of GSV_ACTIVE. The operation transitions CXTV_STOP_FN and CXTV_RUN to CXTV_RUN for valid contexts; it shall skip and ***not*** log an error if the target context is not in one of the above states or the context is invalid. Note, that DSC_CXT_START_RS shall skip contexts in the CXTV_STOP_SW state.

For each context that the function evaluates for a start operation, the function executes the following ordered set of steps:

1. Performs the Context Valid check (ChkValid:Cxt) described in *"4.3.2, Check Valid Context"*. If the check fails, the function shall not modify CXT_STS.state, skip the remaining steps and further evaluation of the context, and take further action based on the type of start operation being executed and the ChkValid:Cxt failure signature.
    - For DSC_CXT_START_NM with any ChkValid:Cxt failure signature, log an error.
    - For DSC_CXT_START_RS with a ChkValid:Cxt failure signature of LogErr:Cxt, log an error.
    - For DSC_CXT_START_RS with a ChkValid:Cxt failure signature of Invalid:Cxt, do not log an error.
2. Using the value of CXT_STS.state obtained in the previous step, if the value of CXT_STS.state is neither CXTV_RUN nor CXTV_STOP_[SW, FN], the function shall perform the following actions based on the type of start operation being executed.
    - For DSC_CXT_START_NM, log an error, do not modify CXT_STS.state, and skip the remaining steps and further evaluation of the context.
    - For DSC_CXT_START_RS, do not log an error, do not modify CXT_STS.state, and skip the remaining steps and further evaluation of the context.
3. For DSC_CXT_START_RS, if CXT_STS.state is CXTV_STOP_SW, do not log an error, do not modify CXT_STS.state, and skip the remaining steps and further evaluation of the context.
4. At this point, if CXT_STS.state is either CXTV_RUN or CXTV_STOP_[SW, FN]:
    - The function will issue an atomic write of the CXT_STS.state to CXTV_RUN.
    - The function will then ensure that the value of CXT_STS.state is CXTV_RUN. If it is not, the function shall log an error. The function may use any implementation-specific mechanism to ensure this including an atomic read back of CXT_STS.state.
5. If the modification of CXT_STS.state to CXTV_RUN is successful, the context is started (transitioned to the CXTV_RUN state).

If all target contexts have been started or skipped successfully without error, these operations return a successful completion; otherwise, they return failure and log errors. See *"5.5, Descriptor Completion"* for more details.

In the case of successful completion and only after the completion status block has been written, these operations are required to evaluate a successfully started context's descriptor ring for new descriptors only if a doorbell_value is specified in the start descriptor. This is done in the descriptor by setting "dv" and placing a value in the doorbell_value field. See *"4.3.3, Doorbell Register and Context Signaling"* for how doorbell_value is used. The function is not required to evaluate a started context's descriptor ring when "dv" is not set.

In the case where a start operation has failed, idempotency is ensured; the operation may be re-submitted several times with the same input without changing the final result beyond its first execution until the operation succeeds. This may be useful if an operation targeting multiple contexts has failed and needs to be resubmitted after remediation.

Figure 6-12: DSC_CXT_START_[NM, RS] Descriptor Format

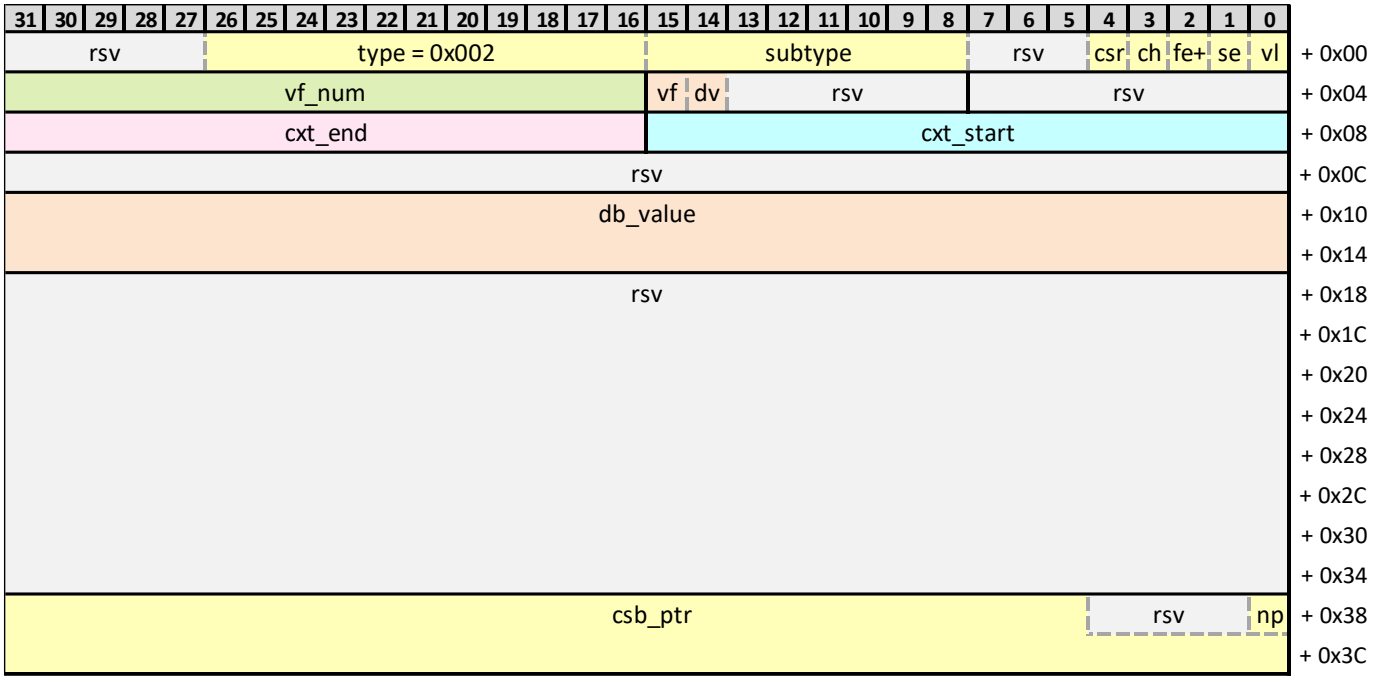


Table 6–14: DSC_CXT_START[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = 1	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype= NM:0x03, RS:0x08	
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 rsvd_0;	039:032		Shall be set to 0
u8 vflags;	045:040	rsvd	Shall be set to 0
	46	dv	doorbell_value valid. When set to 1, after starting the context, the function evaluates the context's descriptor ring for new descriptors using the doorbell_value field.
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u16 cxt_start;	079:064		Indicates the starting context number to operate on.
u16 cxt_end;	095:080		Indicates the ending (last) context number to operate on.
u8 rsvd_1[4];	127:096		Shall be set to 0
u64 db_value;	191:128		64-bit doorbell_value used when "dv" is "1".
u8 rsvd_2[32];	447:192		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.4 AdminGrp DSC_CXT_STOP Operation

The DSC_CXT_STOP operation initiates background actions that stop all target contexts from processing new descriptors and changes each target context's state to a relevant final stopped state. The target context shall be stopped at a descriptor boundary. See *"4.3.5, Function and Context Stop Actions" for m*ore details. The targeted contexts are described using the cxt_start and cxt_end fields. The DSC_CXT_STOP operation shall complete in a timely manner so that further administrative operations may be executed.

The SDXI function shall signal completion of the operation when it determines whether the required background stopping actions can be initiated without error. The operation's completion does not indicate the success nor failure of the background stopping actions for the target context(s) nor does it imply whether the stopping actions have begun or finished.

When the stopping actions finish for a valid context, the CXT_STS.state value shall be either: CXTV_STOPG_[SW, FN]; or CXTV_ERR_FN if a background stopping action encounters an error or abort in stopping a context. If CXTV_ERR_FN is returned, an error shall be logged. The stopping actions initiated by this operation ignore invalid contexts (CXTV_INVALID).

Software may determine stop completion by one of these methods:

- Submit a DSC_SYNC.STOP operation and wait for its completion.
- Check each target context's CXT_STS.state to determine whether the target context is at CXTV_STOP_[SW, FN] or CXTV_ERR_FN.

The DSC_CXT_STOP operation specifies a context completion wait policy using the Hard Stop (HS) bit in the descriptor. When HS = 0, the function shall perform a soft-stop completion wait. When HS = 1, the function shall perform a hard-stop completion wait. See *"4.3.5, Function and Context Stop Actions"* for details.

Multiple DSC_CXT_STOP operations may be submitted against the same context regardless of the value of its CXT_STS.state and the progress of previous stop actions upon it. This allows software to initiate a subsequent hard-stop completion if a previous soft-stop completion is not making acceptable forward progress.

But note that if a DSC_SYNC.STOP operation is issued between two DSC_CXT_STOP operations targeting the same context, the second DSC_CXT_STOP shall have no effect on the stopping actions initiated by the first DSC_CXT_STOP operation.

In the case where the DSC_CXT_STOP operation has failed, idempotency is ensured; the operation may be re-submitted several times with the same input without changing the final result beyond its first execution until the operation succeeds. This may be useful if an operation targeting multiple contexts has failed and needs to be resubmitted after remediation.

From the time the operation is submitted until the time that the CXT_STS.state of a targeted valid context becomes CXTV_STOP_[SW, FN] or CXTV_ERR_FN, software shall not modify any memory-based data structure encompassed by LVL_CXT_CTL. Doing so may cause the context to enter undefined operation. (See *"4.3.1.2, Software Procedure For Modifying Memory-Based Data Structures"*)

Note that this operation is fenced with respect to prior descriptor operations on the same administrative context. See *"6.1.1, Common Header and Footer"* for details.

Figure 6-13: DSC_CXT_STOP Descriptor Format

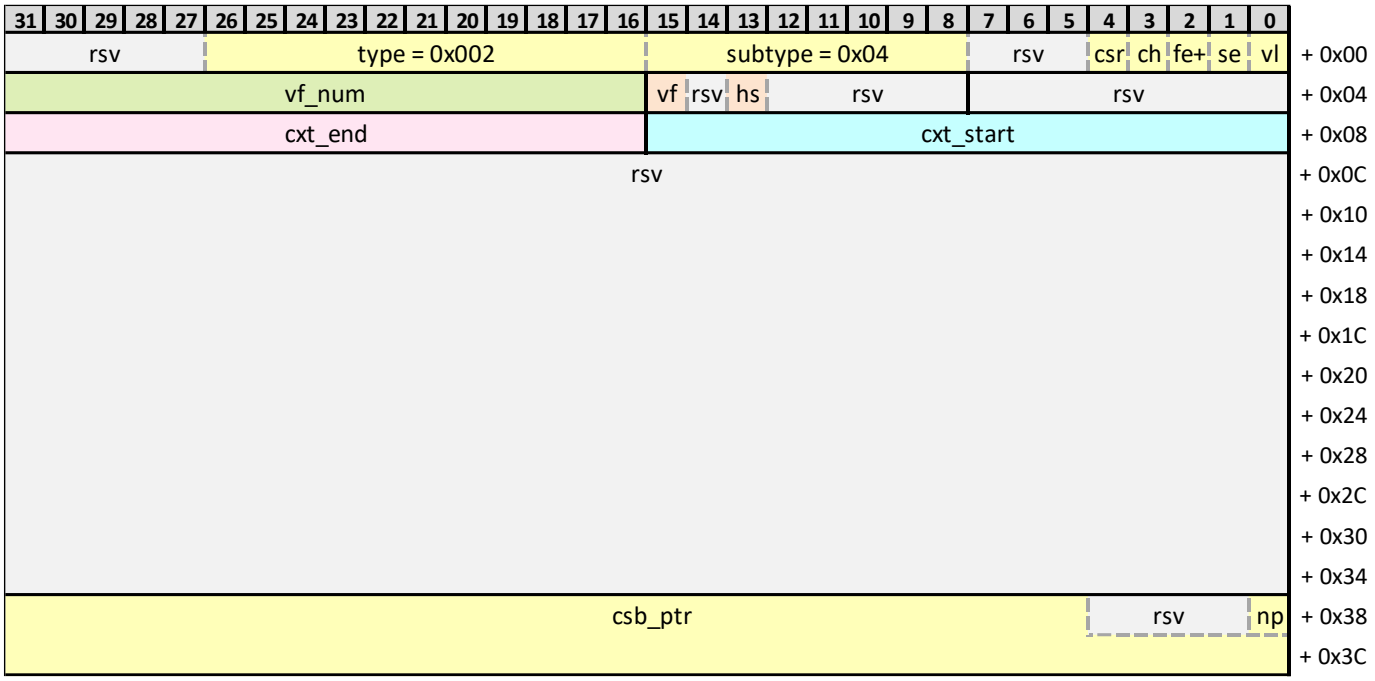


Table 6–15: DSC_CXT_STOP[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = 1	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x04	See section *"6.1.1".*
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 rsvd_0;	039:032		Shall be set to 0
u8 vflags;	044:040	rsvd	Shall be set to 0
	045	hs	0 = Soft-Stop Completion. 1 = Hard Stop Completion.
	46	rsvd	Shall be set to 0
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u16 cxt_start;	079:064		Indicates the starting context number to operate on.
u16 cxt_end;	095:080		Indicates the ending (last) context number to operate on.
u8 rsvd_1[44];	447:096		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.5 AdminGrp DSC_AKEY_UPD Operation

The AdminGrp's DSC_AKEY_UPD operation is used by software to indicate a change to one or more AKey table entries associated with one or more contexts of a target function. This operation signals the target SDXI function to update its internal copies, if any, of the selected AKey table entries. (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*")

The affected contexts are described using the cxt_start and cxt_end fields. The affected AKey table indices are selected using the akey_start and akey_end fields. Note that a successful completion of a DSC_AKEY_UPD operation only indicates that background update actions shall be started; it does not indicate that the update actions have completed nor that their effects are visible. Software shall use the DSC_SYNC operation to ensure that the background update actions have completed.

Figure 6-14: DSC_AKEY_UPD Descriptor Format

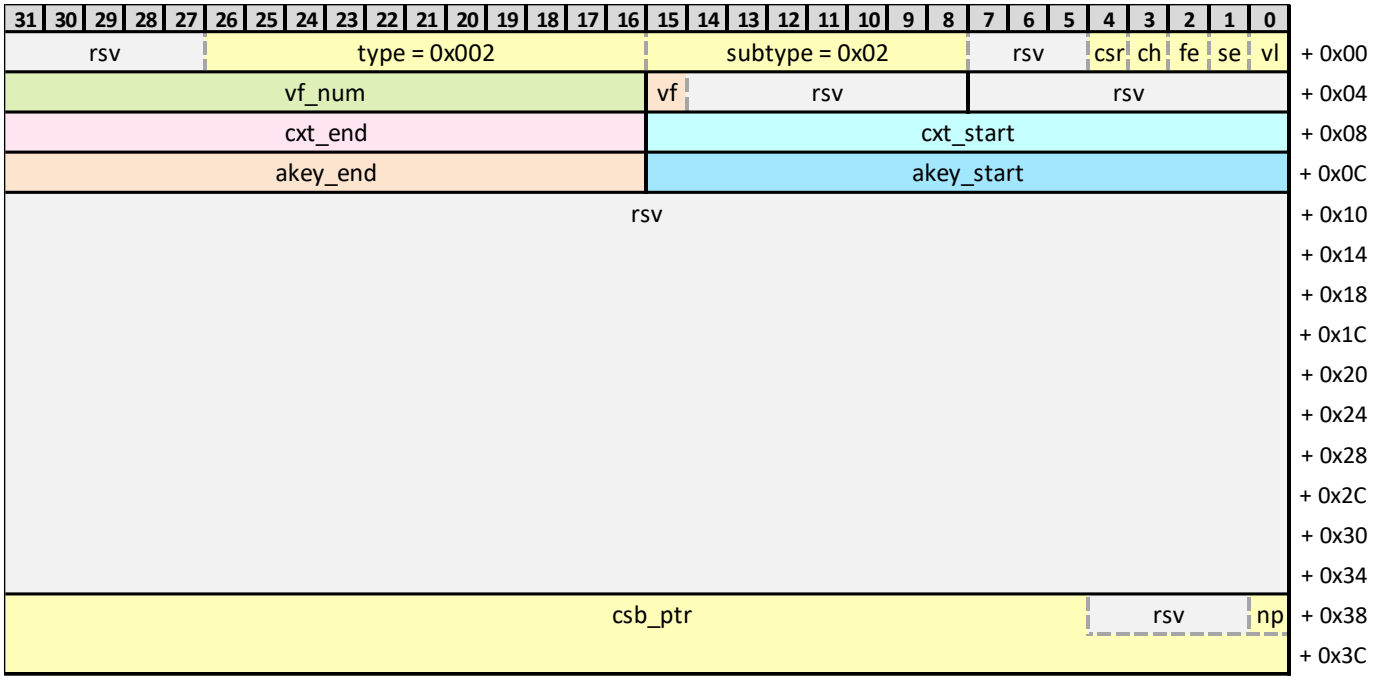


Table 6–16: DSC_ADM_AKEY_UPD[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x02	See section *"6.1.1".*
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 rsvd_0;	039:032		Shall be set to 0
u8 vflags;	046:040	rsvd	Shall be set to 0
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u16 cxt_start;	079:064		Indicates the starting context number to operate on.
u16 cxt_end;	095:080		Indicates the ending (last) context number to operate on.
u16 akey_start;	111:096		Indicates the starting AKey number to operate on.
u16 akey_end;	127:112		Indicates the ending (last) AKey number to operate on.
u8 rsvd_1[40];	447:128		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.6 AdminGrp DSC_CXT_UPD Operation

The AdminGrp's DSC_CXT_UPD operation is used by software to indicate a change to memory data structures describing one or more contexts of a target function. For each context specified, the operation signals the function to start the updating of all internal copies, if any, of all memory-based data structures and subsidiary structures associated with the context at the specified data-structure hierarchy level. (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*".) The targeted contexts are described using the cxt_start and cxt_end fields. The data-structure level (dsl) field within the descriptor determines whether the LVL_L2, LVL_L1, or LVL_CXT_CTL level associated with the context is updated.

When selecting LVL_L2, software shall specify cxt_start as the first context number mapped to the CXT_L2_ENT and cxt_end as the last function-supported context mapped to the CXT_L2_ENT. This is because one CXT_L2_ENT points to a naturally-aligned range of 128 contexts. For example, assume that the function has been configured through MMIO_CAP1 and MMIO_CTL2 to support contexts 0 to 257. Updating the CXT_L2_ENT that maps context numbers 128 to 255, requires a cxt_start of 128 and a cxt_end of 255. Whereas, updating the CXT_L2_ENT that maps context numbers 256 and 257 requires a cxt_start of 256 and a cxt_end of 257.

The function shall not return an error when performing an update on a context that is CXTV_INVALID.

For any requested level of the memory-based data-structure hierarchy to update, an SDXI implementation may update any associated higher level of the hierarchy. Note that a successful completion of a DSC_CXT_UPD operation only indicates that background update actions shall be started; it does not indicate that the update actions have completed nor that their effects are visible. Software shall use the DSC_SYNC operation to ensure that the background update actions have completed.

Table 6–17: Targeted Data-Structure Level

dsl[2:0]	Targeted Data-Structure Level (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*")
000b	Reserved
100b	DSC_CXT_UPD.CTL: Updating LVL_CXT_CTL fields.
110b	DSC_CXT_UPD.L1: Updating LVL_L1 fields.
111b	DSC_CXT_UPD.L2: Updating LVL_L2 fields.
All other encodings are reserved.	

Figure 6-15: DSC_CXT_UPD Descriptor Format

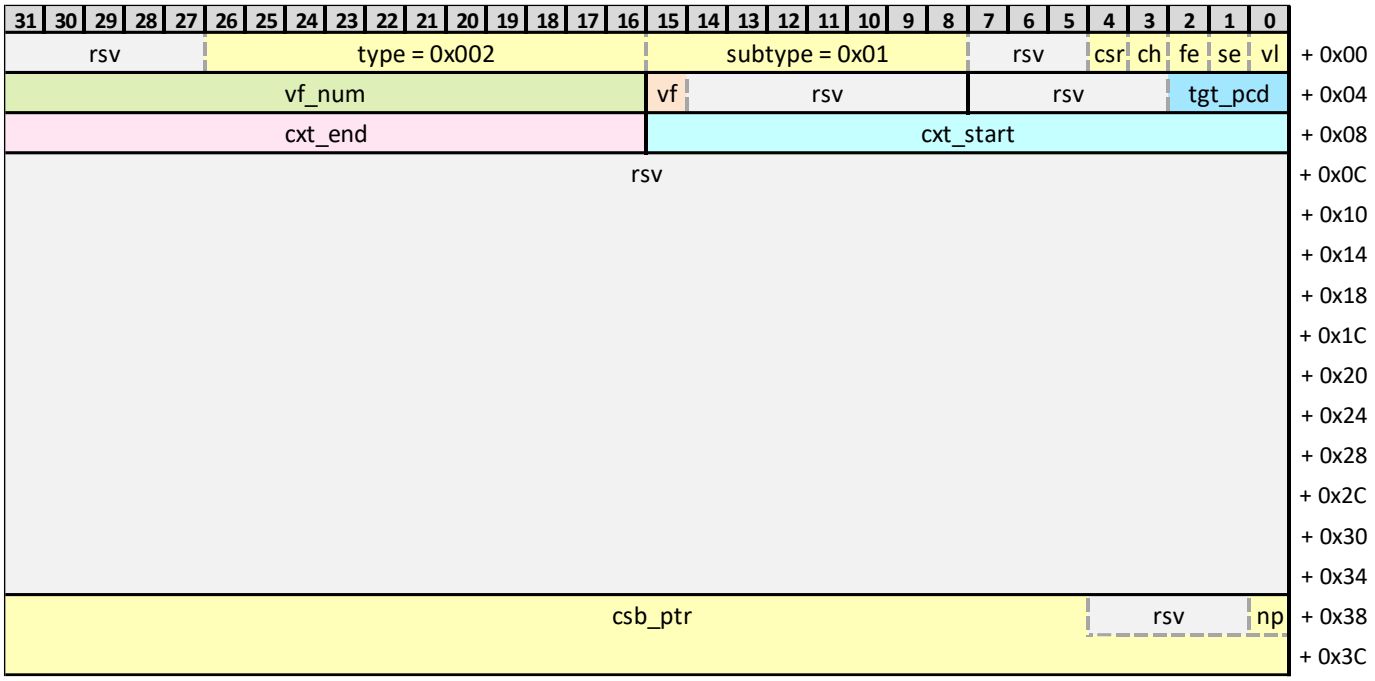


Table 6–18: DSC_CXT_UPD[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x01	See section *"6.1.1".*
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 cflags;	034:032	tgt_pcd	Targeted Private-Cached Data to Update. See *"Table 6–17: Targeted Data-Structure* Level*"* for details.
	039:035	rsvd	Shall be set to 0
u8 vflags;	046:040	rsvd	Shall be set to 0
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u16 cxt_start;	079:064		Indicates the starting context number to operate on.
u16 cxt_end;	095:080		Indicates the ending (last) context number to operate on.
u8 rsvd_0[44];	447:096		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.7 AdminGrp: DSC_FN_UPD Operation

The AdminGrp's DSC_FN_UPD operation is used by software to indicate a non-specific change in memory data structure contents. This operation signals the target SDXI function to invalidate all non-coherently cached copies of SDXI data structures across all contexts (LVL_FN).

Note that a successful completion of a DSC_FN_UPD operation only indicates that background update actions shall be started; it does not indicate that the update actions have completed nor that their effects are visible. Software shall use the DSC_SYNC operation to ensure that the background update actions have completed.

Figure 6-16: DSC_FN_UPD Descriptor Format

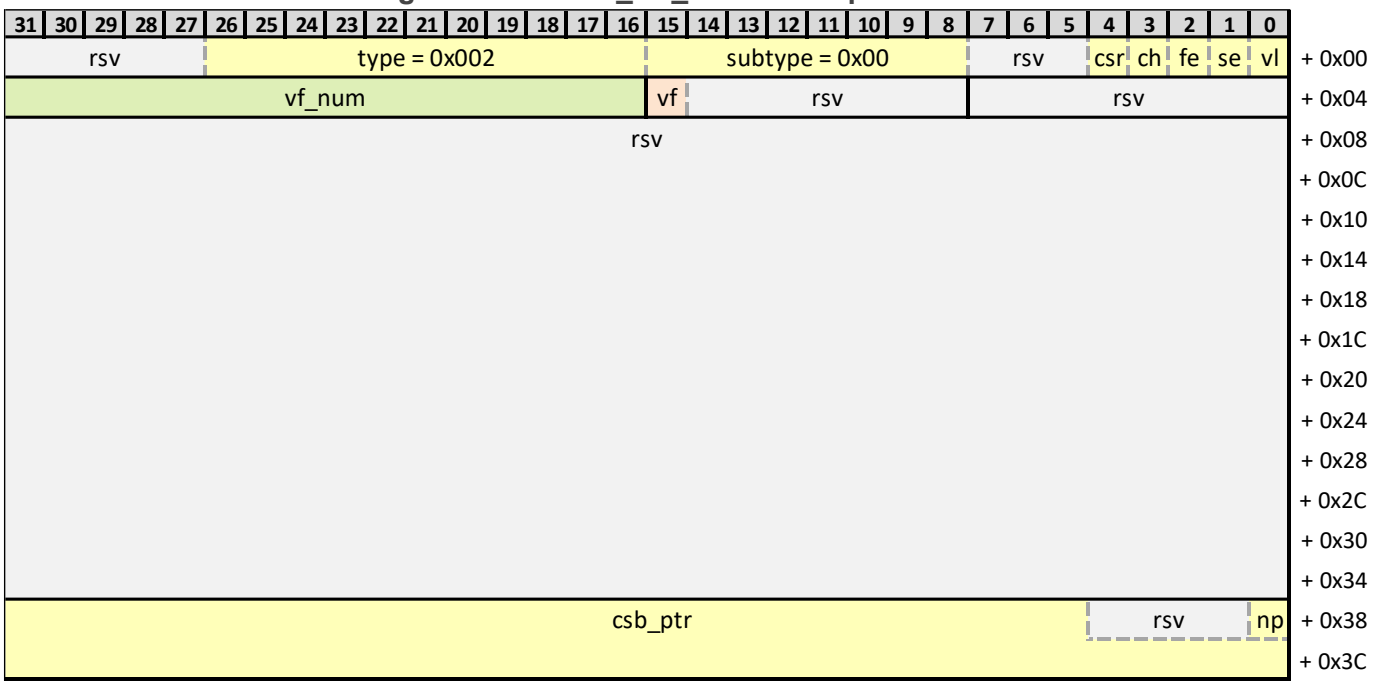


Table 6–19: DSC_FN_UPD[^7] Operation Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x00	See section *"6.1.1".*
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 rsvd_0;	039:032	rsvd	Shall be set to 0
u8 vflags;	046:040	rsvd	Shall be set to 0
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u8 rsvd_1[48];	447:064		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.8 AdminGrp DSC_RKEY_UPD Operation

The AdminGrp's DSC_RKEY_UPD operation is used by software to indicate a change to one or more RKey table entries. This operation signals the target SDXI function to update its internal copies, if any, of the selected RKey table entries. (See *"4.3.1, SDXI Memory-Based Data-Structure Hierarchy and Caching*") The affected RKey table indices are selected using the rkey_start and rkey_end fields.

Note that a successful completion of a DSC_RKEY_UPD operation only indicates that background update actions shall be started; it does not indicate that the update actions have completed nor that their effects are visible. Software shall use the DSC_SYNC operation to ensure that the background update actions have completed.

Figure 6-17: DSC_RKEY_UPD Descriptor Format

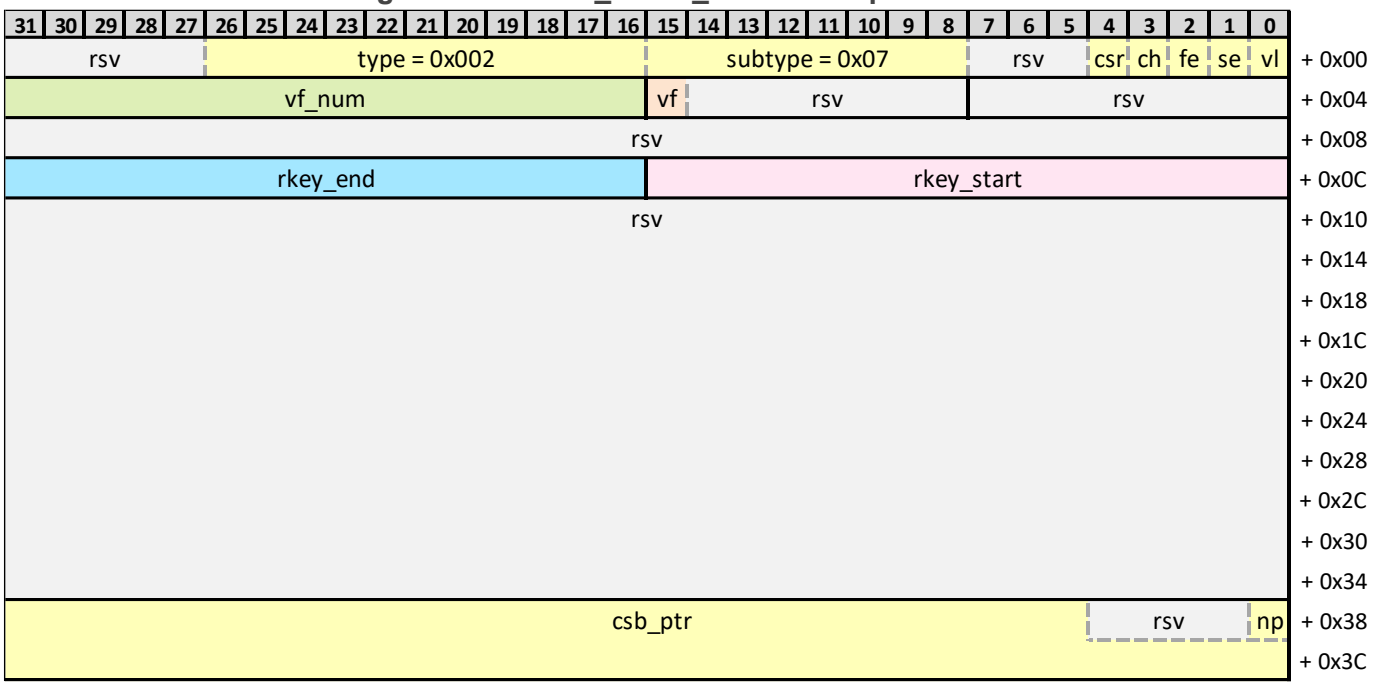


Table 6–20: DSC_RKEY_UPD[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x07	See section *"6.1.1".*
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 rsvd_0;	039:032	rsvd	Shall be set to 0
u8 vflags;	046:040	rsvd	Shall be set to 0
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u8 rsvd_1[4];	095:064	rsvd	Shall be set to 0
u16 rkey_start;	111:096		Indicates the starting RKey number to operate on.
u16 rkey_end;	127:112		Indicates the ending (last) RKey number to operate on.
u8 rsvd_2[40];	447:128		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.9 AdminGrp DSC_SYNC Operation

The AdminGrp's DSC_SYNC operation acts as a filtered synchronization barrier for prior DSC_UPD operations, DSC_CXT_STOP operations, and background stopping actions targeting a specific function. The filter field in the DSC_SYNC descriptor specifies which data structures updates and background actions are ordered with respect to the synchronization barrier for the targeted function.

A DSC_SYNC operation is issued from the administrative context of a source function and synchronizes against the specified operations and actions previously applied to a target function. When the "vf" field of the operation is "0", both the source and target are the same local function (referred to as a "LOC" operation). When the "vf" field is "1", the source function is a PF and the target function is a VF specified by the "vf_number" field (referred to as a "P2V" operation).

The DSC_SYNC synchronization barrier is applied as follows.

1. For all DSC_SYNC filters except STOP, a DSC_SYNC operation shall ensure synchronization with previous DSC_UPD operations that match the filter and are issued from the same source function acting upon the same target function. Consider the following examples where the filter matches.
    a. A DSC_SYNC operation is ensured to synchronize with a previous DSC_UPD when both operations have "vf" as "0" and are issued on the same function.
    b. A DSC_SYNC operation is ensured to synchronize with a previous DSC_UPD when both operations have "vf" as "1", are issued on the same PF, and target the same "vf_number".
    c. A DSC_SYNC operation is not ensured to synchronize with a previous DSC_UPD when one operation is issued by the PF targeting a VF ("vf" is "1") and the other has been locally issued ("vf" is "0") on the same VF.
    d. A DSC_SYNC operation is not ensured to synchronize with a previous DSC_UPD when both operations have "vf" as "1", are issued on the same PF, but target a different "vf_number".
2. For a DSC_SYNC.STOP, the operation shall ensure synchronization with the stopping of the specified contexts on the target function – regardless of the source of the stopping action. Consider the following examples where the filter matches.
    a. A DSC_SYNC.STOP operation with "vf" as "0" is ensured to synchronize with the stopping of a context on the same function.
    b. A DSC_SYNC.STOP operation issued by the PF targeting a context on a VF ("vf" is "1") is ensured to synchronize with the local stopping of the same context on the same VF.
    c. A DSC_SYNC.STOP operation issued by the PF targeting a context on a VF ("vf" is "1") is not ensured to synchronize with the stopping of any context on the PF or a different VF.
3. For any specified filter and target function, an SDXI implementation may synchronize against any larger set of updates, actions, and functions that includes the specified filter and target function; software shall not rely upon this behavior.

A successful completion of the DSC_SYNC operation indicates the successful completion of the specified data structures updates and background actions, and all previously received MMIO doorbells are fully evaluated (i.e. there are no outstanding data structure reads due to a prior doorbell). An error associated with DSC_SYNC may indicate an error associated with the same updates and background actions.

The table below describes the filter field used by DSC_SYNC.

Table 6–21: DSC_SYNC Filter

Filter	Name	Description
000b	CXT	The operation synchronizes against context updates (DSC_CXT_UPD.L2, DSC_CXT_UPD.L1, DSC_CXT_UPD.CTL, and DSC_CXT_UPD.AKEY) for the contexts specified by the cxt_start and cxt_end fields. The key_start and key_end fields are ignored.
001b	STOP	The operation synchronizes against completed stop actions on the target function (DSC_CXT_STOP, StopErr:Cxt, and Stop_FN) for the contexts specified by the cxt_start and cxt_end fields. The key_start and key_end fields are ignored. Note that synchronizing against all function contexts for stopping is insufficient to determine that the function itself has transitioned to GSV_STOP; software must still check MMIO_STS0.fn_gsv for GSV_STOP.
010b	AKEY	The operation synchronizes against AKey updates (DSC_AKEY_UPD) for the contexts specified by the cxt_start and cxt_end fields and the AKeys specified by the key_start and key_end fields.
011b	RKEY	The operation synchronizes against RKey updates (DSC_RKEY_UPD) for the the RKeys specified by the key_start and key_end fields. The cxt_start and cxt_end fields are ignored.
100b	Function (FN)	The operation synchronizes against function updates (DSC_FN_UPD). The cxt_start, cxt_end, key_start, and key_end fields are ignored.
All other values are reserved.		

Figure 6-18: DSC_SYNC Descriptor Format

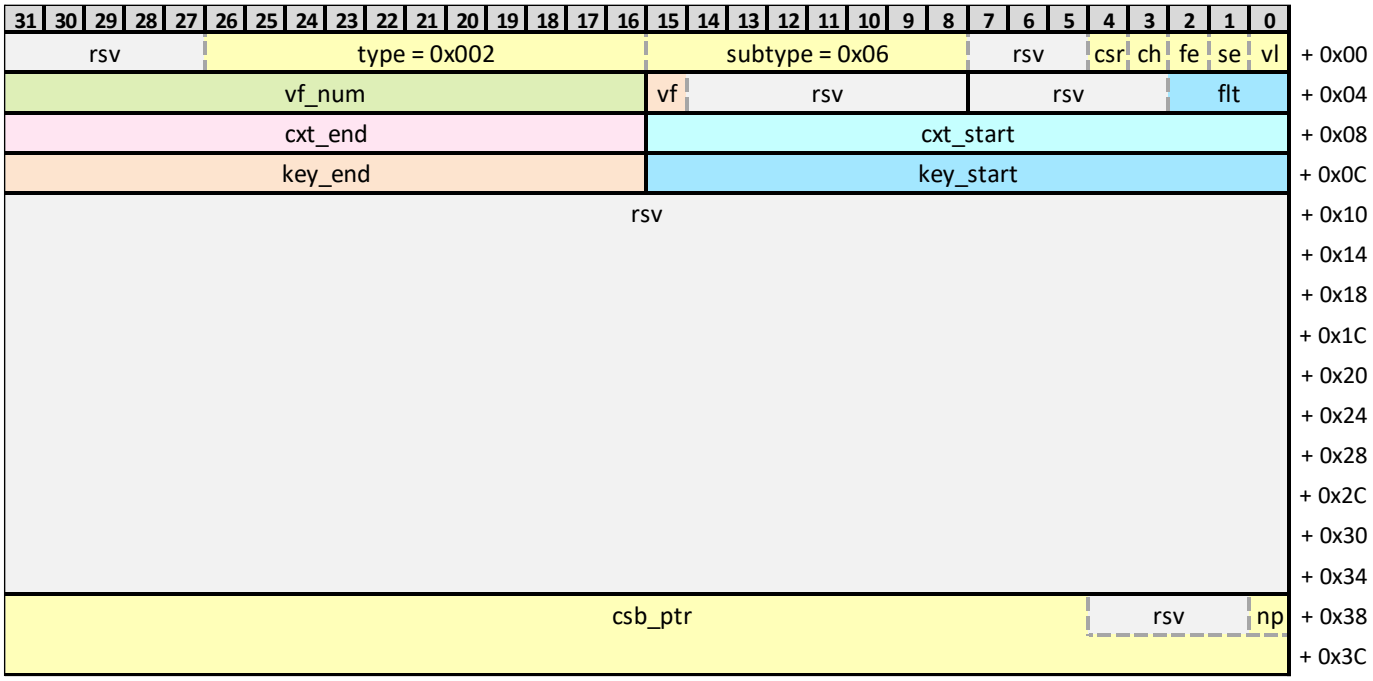


Table 6–22: DSC_SYNC[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1".*
	001	se = *	Sequential Consistency. See section *"6.1.1".* Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1".*
	003	ch = 0	Chain. See section *"6.1.1".* Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x06	See section *"6.1.1".*
	026:016	type=0x002	See section *"6.1.1".*
	031:027	rsvd	Shall be set to 0
u8 cflags;	034:032	flt	Indicates the filter to apply.
	039:035	rsvd	Shall be set to 0
u8 vflags;	046:040	rsvd	Shall be set to 0
	47	vf	"vf" valid. 1 = Update the VF number indicated by vf_number 0 = Update within the local function context executing this descriptor (For VF administrative contexts, this field shall be set to 0.)
u16 vf_num;	063:048		"vf" number. Only valid for PF administrative contexts. When VF=1, this field indicates the VF to update. (For VF administrative contexts, this field shall be set to 0.)
u16 cxt_start;	079:064		Indicates the starting context number to operate on.
u16 cxt_end;	095:080		Indicates the ending (last) context number to operate on.
u16 key_start;	111:096		Indicates the starting key number to operate on.
u16 key_end;	127:112		Indicates the ending (last) key number to operate on.
u8 rsvd_0[40];	447:128		Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1".*
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1".*

6.6.10 AdminGrp DSC_ADM_INTR Operation

The AdminGrp's Interrupt operation is used to generate interrupts from within an administrative context. The interrupt is generated from the local function hosting the administrative context.

Figure 6-19: DSC_ADM_INTR Descriptor Format

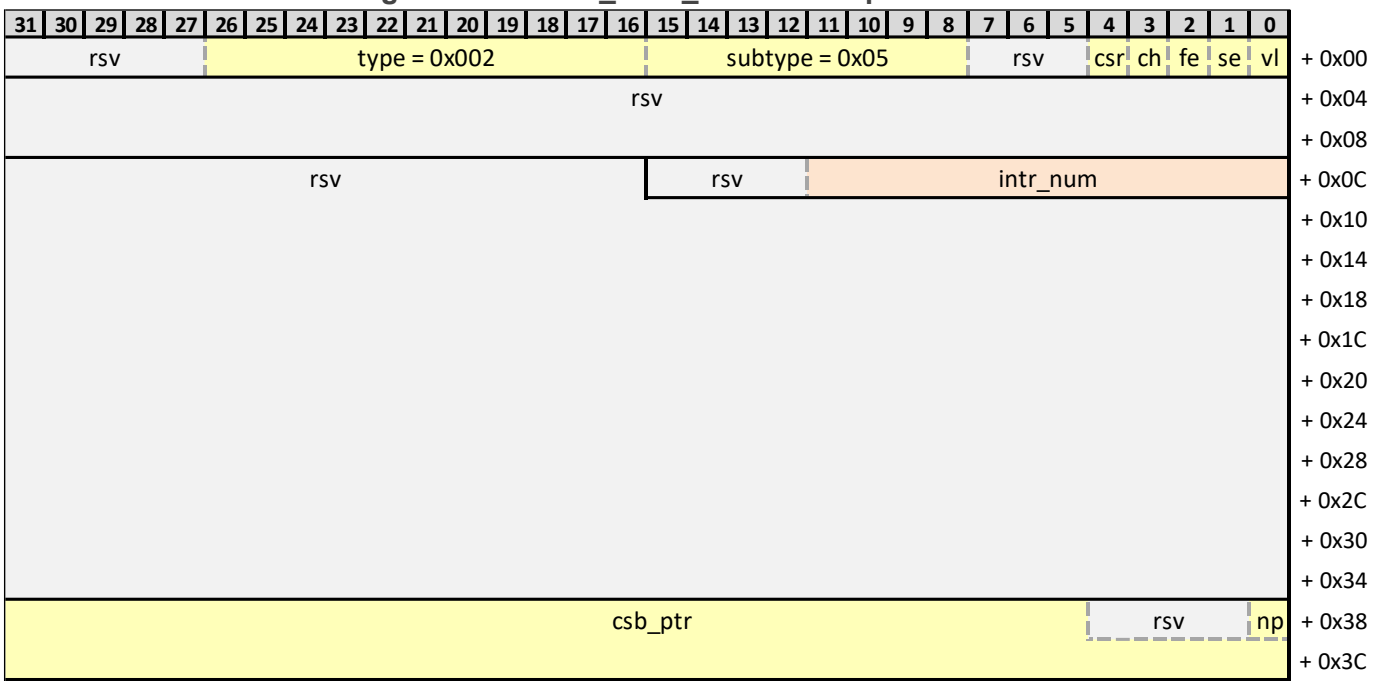


Table 6–23: DSC_ADM_INTR[^7] Descriptor Format

Field	Bits	Subfield	Description
u32 opcode;	000	vl = 1	Valid. See section *"6.1.1"*.
	001	se = *	Sequential Consistency. See section *"6.1.1"*. Shall be set to 0.
	002	fe = *	Fence. See section *"6.1.1"*.
	003	ch = 0	Chain. See section *"6.1.1"*. Shall be set to 0.
	004	csr	Completion Status Mode Requirement for the descriptor.
	007:005	rsvd	Shall be set to 0
	015:008	subtype=0x05	See section *"6.1.1"*.
	026:016	type=0x002	See section *"6.1.1"*.
	031:027	rsvd	Shall be set to 0
u8 rsvd_0[8];	095:032		Shall be set to 0
u16 intr_num;	107:096	intr_num	Specifies the MSI-X entry used to generate the interrupt.
	111:108	rsvd	Shall be set to 0
u8 rsvd_1[42];	447:112	"	Shall be set to 0
u64 csb_ptr;	448	np	no_pointer. See section *"6.1.1"*.
	452:449	rsvd	Shall be set to 0
	511:453	csb_ptr	Completion Status Block pointer. See section *"6.1.1"*.

7 Recommended Sequences for Function Management

7.1 Function Level Resources

7.1.1 Context Level 2 Table Base (MMIO_CXT_L2) Modification

The simplest method is shown below; others are possible.

1. Stop the function using the procedure described in *"4.1.9, Stopping of the SDXI Function by Software"*.
2. Write MMIO_CXT_L2 with the new location of the Context Level 2 Table.
3. Activate the function using the procedure described in *"4.1.8, Activation of the SDXI Function by Software"*.

7.2 Context Level Resources

7.2.1 Context Level 2 Table Entry (CXT_L2_ENT) Modification

Software may modify a Context Level 2 Table Entry (CXT_L2_ENT) for a number of reasons: make the entry invalid, add a Context Level 1 Table, or delete the Context Level 1 Table. The method for any of these cases is essentially the same.

1. Assumptions
    a. Let "cl2_idx" be the index offset of the desired CXT_L2_ENT from the beginning of the context level 2 table; for example, the fourth entry of the context level 2 table has an index of 3.
    b. The desired CXT_L2_ENT points to a context level 1 table of 128 CXT_L1_ENT entries and indirectly to their associated contexts. Therefore, regardless of the individual context state, it is recommended to operate on all of them.
2. If there are running contexts associated with the CXT_L2_ENT, software shall stop them before making any other changes to the CXT_L2_ENT.
    a. If context 0 (the administrative context) is included in the range of contexts associated with the desired CXT_L2_ENT, then for the below steps "start" = 1; otherwise "start" = (cl2_idx << 7).
    b. Issue DSC_CXT_STOP with cxt_start = start and cxt_end = (cl2_idx << 7) + 127.
    c. Issue DSC_SYNC.STOP with cxt_start = start and cxt_end = (cl2_idx << 7) + 127.
    d. Wait for the DSC_SYNC.STOP to complete.
    e. If context 0 is included, issue DSC_CXT_STOP with cxt_start = 0 and cxt_end = 0. Wait until CXT_STS.state of Context 0 is STOP_CXT_SW.
3. Modify the CXT_L2_ENT as appropriate.
    a. If adding a new context level 1 table, initialize the associated CXT_L1_ENTs, and contexts as appropriate.
4. If context 0 is included, it must be restarted first through a jump-start method described in *"4.3.4, Starting A Context and Context Signaling"*.
5. Propagate and synchronize the update.
    a. Issue DSC_CXT_UPD.L2 with cxt_start = (cl2_idx << 7) and cxt_end = (cl2_idx << 7) + 127;
    b. Issue DSC_SYNC.CXT_UPD with cxt_start = (cl2_idx << 7) and cxt_end = (cl2_idx << 7) + 127.
    c. Wait for the DSC_SYNC.CXT_UPD to complete.
6. Restart all relevant contexts if appropriate. All 128 contexts can be started by issuing a DSC_START_NM with cxt_start = (cl2_idx << 7) and cxt_end = (cl2_idx << 7) + 127.

8 SDXI PCI-Express Device Architecture

SDXI is based on the PCI-Express device and software architecture including the use of configuration and MMIO register spaces and standard PCI-Express capabilities including, but not limited to, PCI Power Management, MSI/MSI-X interrupts and SR-IOV.

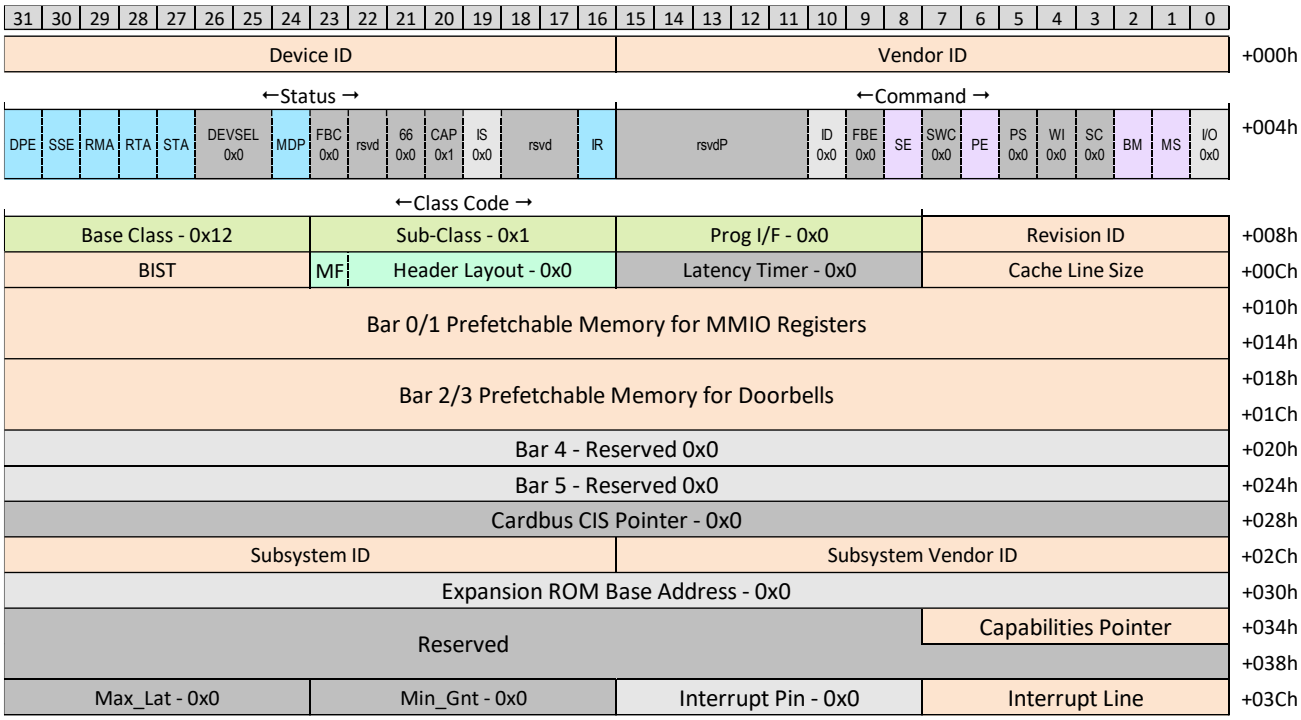
SDXI may be implemented using a range of function types such as PCIe EP or RCiEP. It may be implemented as either a Physical Function or Virtual Function.

Each SDXI function shall support standard PCI-Express defined reset mechanisms such as fundamental reset and hot-reset.

If SR-IOV is supported, PFs and VFs shall support Function Level Reset (FLR).

8.1 SDXI Function Configuration Space Registers

Figure 8-1: PCI Config Space



8.1.1 Class Code

SDXI functions are expected to be identified through the SDXI class code.

- ***SNIA Smart Data Accelerator Interface (SDXI) controller:***
  - Base Class = 0x12
  - Sub Class = 0x01
  - Programming Interface = 0x0

8.1.2 BAR Configuration

Table 8–1: BAR Configuration

PCI Bar Index	Width	Size	Type	Description
Bar 0/1	64 bit	512KiB to 4MiB for PF. 512KiB for VF	Prefetchable, Memory	MMIO Registers
Bar 2/3	64 bit	Variable	Prefetchable, Memory	Doorbell BAR
Bar 4				Reserved
Bar 5				Reserved

If MSI-X is supported, the MSI-X table and PBA Offset table may be placed within any of the BARs including the reserved ones depending on hardware implementation. A block of MMIO space within BAR0/1 is reserved for use by MSI-X tables. Implementations are not required to use this reserved MMIO region.

The MMIO register BAR is marked as prefetchable to allow software to allocate the region within 64-bit MMIO space in order to avoid exhaustion of 32-bit MMIO space when implementing SR-IOV with many virtual functions. Please refer to the implementation note in the PCI-Express Base Specification which provides additional guidance on the use of the prefetchable attribute.

8.1.3 Required Capabilities and Extended Capabilities

SDXI functions are required to support the following capabilities:

- PCI Power Management
- MSI and/or MSI-X

Additional PCIe capabilities such as Advanced Error Reporting, Transaction Processing Hints, and Single Root I/O Virtualization are optional. See "*4.1.7, Function Reset and Outstanding DMA Requests"* for reset requirements.

8.2 Mapping sfunc Values to PCIe Requester ID Values

PCIe SDXI Functions map sfunc values onto PCIe Requester ID values. The mapping is implementation specific. Software may determine the mapping by reading MMIO_CAP0.sfunc and associating the value with the Requester ID of the PCIe Function owning the register.

8.3 Mapping SDXI DSH to PCIe TLP Processing Hints (PCIe TPH)

A PCIe SDXI Function may support PCIe TLP Processing Hints (PCIe TPH) and map DSH information onto the corresponding PCIe TPH fields when accessing data buffers over the PCIe link. See "*3.6, Data Steering Hints (DSH)*" for more details on DSH.

A PCIe SDXI Function indicates support for PCIe TPH functionality through the inclusion of the PCIe TPH Requester Extended Capability. PCIe TPH capabilities such as the supported steering tag modes are indicated through the TPH Requester Capability Register.

System software is responsible for selecting the desired steering tag mode via the ST Mode Select register and enabling PCIe TPH by setting the PCIe TPH Requester Enable register to 1.

When a data buffer access has DSH information included, and PCIe TPH is enabled, the corresponding request on the PCIe link shall be generated with the PCIe TH (Transaction Hint) field set to 1. The PCIe TPH PH (Processing Hint) field is directly copied from the DSH "ph" field obtained from the data buffer's AKey table entry (AKEY_ENT.ph) or RKey table entry (RKEY_ENT.ph) as appropriate. The PCIe TPH ST (Steering Tag) information is controlled, in part, by the PCIe-defined ST Mode Select register:

- If No ST Mode is enabled, ST is set to 0.
- If Interrupt Vector Mode is enabled, the ST value is obtained by using the intr_num (PCIe interrupt vector number) to index into the appropriate PCIe-defined structure holding the ST values, such as the MSI-X table. The intr_num is obtained from the data buffer's AKey table entry or, if present, RKey table entry (AKEY_ENT/RKEY_ENT.intr_num). The table entry must have the "iv" field set to 1 for intr_num to be valid. If intr_num is not valid, then ST is set to 0. The "stag" field in the AKey or RKey table entry is not used.
- If Device Specific Mode is enabled, ST is obtained from the data buffer's AKey table entry (AKEY_ENT.stag) or RKey table entry (RKEY_ENT.stag) as appropriate.

The use and control of PCIe TPH information with non-data-buffer accesses is implementation-specific

Refer to the PCI-Express Base Specification for further details on PCIe TLP Processing Hints.

8.4 PCIe Atomic Capabilities Discovery and Enablement

Privileged software can determine whether a PCIe SDXI Function supports initiating atomic operations through the mechanisms described in 4.4 Atomic Operation Support. To enable a PCIe SDXI Function to initiate atomic operations, privileged software shall set the AtomicOp Requester Enable bit in the Device Control 2 Register. This register is located in the PCIe SDXI Function's configuration space PCI Express Capability structure.

Privileged software can determine whether a PCIe Function supports completing atomic operations by checking the 32-bit AtomicOp and 64-bit AtomicOp supported fields in the Device Capabilities 2 Register of that Function's configuration space PCI Express Capability structure. The Function may be an upstream Root Port for the PCIe SDXI Function, or another PCIe Endpoint Function acting as a peer-to-peer target.

Privileged software can check whether Bridge Functions, such as Switch Port and Root Port, located between a PCIe SDXI Function and a target PCIe Function support routing atomic operations by checking the AtomicOp Routing Supported field in the Device Capabilities 2 Register of those Bridge Functions" configuration space PCI Express Capability structures.

9 MMIO Control Registers

Each function's MMIO registers are grouped by category within separate 64Kbyte regions, as shown in *"Table 9–1: MMIO Control Registers"*, to facilitate CPU MMU-based protection and emulation across a range of possible processor architectures. PFs may implement an MMIO region of between 512KB and 4MB depending on the amount of MMIO space required to hold mailboxes for the number of supported VFs. VFs each consume 512KB of MMIO space.

The requirements to access an SDXI function's MMIO register are described here. Software should not depend on the results of any unsupported access (as described in this section) to these MMIO registers. A misaligned access that straddles MMIO registers is not supported and results in undefined operation of the SDXI function. A write to an MMIO register must be ordered with respect to other accesses to the same register; software must use processor-specific means to ensure that the required memory ordering is enforced.

For PCIe implementations, the SDXI function's MMIO registers are located in PCIe prefetchable memory space. While it is expected that most platforms will not merge separate writes to an SDXI function, software may be required on some platforms to work around this when writing SDXI function MMIO registers. Please consult platform-specific references and the *PCI Express Base Specification Revision 5.0 -- Implementation Note: "Additional Guidance on the Prefetchable Bit in Memory Space BARs"*.

The SDXI function's MMIO register space is segmented logically into two regions for the purposes of discussing the access model: the Doorbell MMIO registers, and the remaining MMIO registers.

For the Doorbell MMIO registers, only a naturally-aligned 64-bit write is supported. Any other write access is ignored by the SDXI function. A read access of a Doorbell MMIO register is not supported; the SDXI function must return a result but the result is architecturally undefined.

For the remaining MMIO registers, the SDXI function supports all naturally-aligned accesses up to 64-bits as atomic with one exception: if an implementation can not support 64-bit atomic accesses it will report MMIO_CAP1.mmio64 as "0". Misaligned reads and writes within an MMIO register are not supported; the SDXI function is allowed to return or write stale data as determined by the access type. Attempting to write reserved fields in an MMIO register with an illegal value results in an undefined value for the register.

A "torn" read or write of an MMIO register can result when at least two agents - any of the SDXI function or one or more software threads -- are concurrently accessing the fields of the register in a non-atomic manner. The torn access can result in stale data being intermixed with more recent data in an unpredictable way. Software is solely responsible for avoiding torn reads and writes on accesses that straddle fields within an MMIO register. Software shall solely ensure synchronized access among multiple software threads; the method for which is beyond the scope of this specification.

In an environment with synchronized software access, the following list describes methods a single software thread can use to avoid undefined operation and torn accesses with respect to the SDXI function itself; SDXI implementations shall ensure that these methods are supported.

- If the register is not dynamically updated by the SDXI function, then any naturally-aligned read will avoid returning torn data.
- When the SDXI function's MMIO_STS0.fn_gsv is "GSV_STOP", any naturally-aligned read or write will avoid torn data.
- This is the only supported method for MMIO_CXT_L2 (MMIO Offset: 0x1_0000)
- If possible, use a supported naturally-aligned atomic read or write that maps solely to complete fields in the MMIO register.

- If the MMIO register has an associated valid/enable bit (in some cases this is in another MMIO register), then disable or invalidate the register, read the enable/valid bit back to confirm it has changed value, make modifications, and then enable or validate the register.

The Error log read and write indexes are 64-bit MMIO registers and SDXI functions that report MMIO_CAP1.mmio64 as “0” may exhibit torn accesses. Since the Error Log can contain no more than 2^26 entries, the contents of the ring can be calculated and read by software using only the lower 32-bits of the index registers, and thereby avoid the effects of torn reads of MMIO_ERR_WRT. An SDXI function that reports MMIO_CAP1.mmio64 as “0” must account for the possibility of receiving torn MMIO writes to MMIO_ERR_RD.

Table 9–1: MMIO Control Registers

64KB Region	PF Usage	VF Usage
0x00_0000	General Control and Status Registers	
0x01_0000	Context Table Registers	
0x02_0000	Error Logging Control and Status Registers	
0x03_0000	Mailbox Control Registers	
0x04_0000	Reserved for MSI-X	
0x05_0000	Reserved	
0x06_0000 to 0x25_FFFF	Alternating 64K regions of Send and Receive Mailboxes	Send Mailbox at 0x6_0000. Receive Mailbox at 0x7_0000.
0x26_0000 to 0x3F_FFFF	Reserved	N/A

9.1 General Control and Status Registers

Table 9–2: MMIO_CTL0[^1] (MMIO Offset :0x0_0000)

Field	Bits	Subfield	Type & Reset	Description
u32 field0;	001:000	fn_gsr	RW, 0x0	Controls the SDXI function state. See "*4.1 SDXI Function State"* for more detail. fn_gsr 0b00 GSRV_RESET 0b01 GSRV_STOP_SF 0b10 GSRV_STOP_HD 0b11 GSRV_ACTIVE
	002	fn_pasid_vl	RW, 0x0	Indicates whether the fn_pasid field is used when accessing context level 2 or 1 table entries, Context Control Entries, AKey table entries, RKey table entries, and error log entries. If fn_pasid_vl is “0”, requests to the referenced data structures are accessed using DMA requests without PASID.
	003	rsvd	R, 0x0	Shall be set to zero.
	004	fn_err_intr_en	RW, 0x0	1 = An interrupt is signaled when the function transitions to the GSV_ERROR state as reported through MMIO_STS0.fn_gsv. When MSI or MSI-X is enabled, message 0 is signaled. 0 = No interrupt is generated
	007:005	rsvd	R, 0x0	Shall be set to zero.
	027:008	fn_pasid	RW, 0x0	Function PASID value.
	031:028	rsvd	R, 0x0	Shall be set to zero.
u32 fn_grp_id;	063:032		RW, 0x0	This field is provided for software to record the function group ID assignment. The field has no effect on the operation of the SDXI function. In a VF, the field is read-only and reflects the value of the associated PF. See "3.3.1, SDXI Function Group" for more details.

Table 9–3: MMIO_GRP_ENUM[^1] (MMIO Offset: 0x0_0008)

Field	Bits	Subfield	Type & Reset	Description
u8 field0;	000	busy	RW, 0x0	When "0", there is no outstanding write propagation of this function's "probe" field to the probe fields of other functions in the function group. When "1", there is an outstanding write propagation of this function's "probe" field to the probe fields of other functions in the function group. This field does not propagate. In a VF, the field is read-only and reflects the value of the associated PF. See "*3.3.1, SDXI Function Group"* for more details.
	001	probe	RW, 0x0	The current value of the probe field. When written, this field propagates to the same field in other PFs within the function group. In a VF, the field is read-only and reflects the value of the associated PF. See "*3.3.1, SDXI Function Group"* for more details.
	007:002	rsvd	R, 0x0	This field always reads back as 0.
u8 rsvd_0[7];	063:008		R, 0x0	This field always reads back as 0.

Table 9–4: MMIO_CTL2[^1] (MMIO Offset: 0x0_0010)

Field	Bits	Subfield	Type & Reset	Description
u16 field0;	003:000	max_buffer	RW	This field controls the maximum data buffer size enabled for use by this function. The field is encoded the same as MMIO_CAP1.max_buffer and must not exceed it in value. Reset: MMIO_CAP1.max_buffer
	011:004	rsvd	R, 0x0	Shall be set to zero.
	015:012	max_akey_sz	RW	This field controls the maximum size of any AKey table useable by any context within this function. The field is encoded the same as MMIO_CAP1.max_akey_sz and must not exceed it in value. Reset: MMIO_CAP1.max_akey_sz
u16 max_cxt;	031:016		RW	This field controls the maximum number of contexts enabled for use by this function. The field is encoded the same MMIO_CAP1.max_cxt and must not exceed it in value. Reset: MMIO_CAP1.max_cxt
u32 opb_000_avl;	063:032		RW, 0x0	Each bit in this field, when set to "1", indicates that a certain descriptor operation group is available for all contexts within the function; when "0" it is not. The bit encoding matches the MMIO_CAP1.opb_000_cap register. For a context to use a supported and available operation group, the group must also be enabled in CXT_L1.opb_000_enb. See *"5.1, Descriptor Operations"* and *"Chapter 6, SDXI Descriptor and Operation Specification"* for more details.

Table 9–5: MMIO_STS0* (MMIO Offset: 0x0_0100)

Field	Bits	Subfield	Type & Reset	Description
u8 field0;	002:000	fn_gsv	R, 0x0	This field describes the overall state of the SDXI function. This register does not indicate the state of any specific context 000b – GSV_STOP 001b – GSV_INIT 010b – GSV_ACTIVE 011b – GSV_STOPG_SF 100b – GSV_STOPG_HD 101b – GSV_ERROR All other encodings reserved. See "4.1 SDXI Function State" for more detail.
	007:003	rsvd	R, 0x0	Shall be set to zero.
u8 rsvd_0[7];	063:008		R, 0x0	Shall be set to zero.

Table 9–6: MMIO_CAP0[^1] (MMIO Offset: 0x0_0200)

Field	Bits	Subfield	Type & Reset	Description
u16 sfunc;	015:000		R	An opaque SDXI Function identifier that maps uniquely to an SDXI function. For example, in a PCIe implementation of SDXI, "sfunc" could map to a PCIe Requester ID. It is referenced by AKey table entries. sfunc is unique for each SDXI function within an SDXI Function Group. This field may be set to 0 for implementations with MMIO_CAP1.rkey_cap set to "0".
u8 field0;	016	vf	R	1 = Indicates a virtual function. Software should use the virtual function programming model including MMIO register layout and mailbox programming. 0 = Indicates a physical function. Software should use the physical function programming model including MMIO register layout and mailbox programming.
	018:017	cs_cap	R, IMPL	Completion-Status Capabilities. See *"4.4.1, Completion-Status Capabilities"* for details.
	019	rsvd	R, 0x0	rsvd
	022:020	db_stride	R, IMPL	Indicates the address stride between doorbell sections. The stride is encoded as 2**(db_stride+12) bytes.
	23	rsvd	R, 0x0	rsvd
u8 max_ds_ring_sz;	028:024	max_sz	R, IMPL	Indicates the maximum size of any descriptor ring for any context within this function. The maximum descriptor ring size that is supported is 2**(max_ds_ring_sz + 16) bytes. Valid max_ds_ring_sz values range from 0 (64 KB or 1K descriptors) to 22 (256 GB or 4G descriptors); all other encodings are reserved. Software shall ensure for all valid contexts that: - Let max_ds = 2**(MMIO_CAP0.max_ds_ring_sz + 10) - Then CXT_CTL.ds_ring_sz <= min((2**32)-1, max_ds)
	031:029	rsvd	R, IMPL	rsvd
u8 max_rkey_sz;	035:032	max_sz	R, IMPL	Indicates the maximum size of the RKey table for this function. The maximum RKey table size that is supported is 2**(12 + max_rkey_sz) bytes. Values of max_rkey_sz greater than 0x8 are reserved.
	039:036	rsvd	R, 0x0	rsvd
u8 rsvd_0[3];	063:040		R, 0x0	rsvd

Table 9–7: MMIO_CAP1[^1] (MMIO Offset: 0x0_0208)

Field	Bits	Subfield	Type & Reset	Description
u16 field0;	003:000	max_buffer	R, IMPL	Indicates the maximum data buffer size supported by this function for operations. The maximum data buffer size that is supported is 2**(max_buffer+21) bytes. Valid max_buffer values range from 0 (2 MiB) to 11 (4 GiB); all other encodings are reserved.
	004	rkey_cap	R, IMPL	Indicates whether RKey functionality is supported. RKey support is required to support data transfers between functions in an SDXI group.
	005	rm	R, IMPL	Memory Registration Required. Reserved for use with the SDXI Connection Operation Group.
	006	mmio64	R, IMPL	When 1, indicates the SDXI Function implements 64-bit atomic access to MMIO registers. When 0, 32-bit is the largest atomic access to MMIO registers. See the beginning of "9 MMIO Control Registers" for more details. Behaviors associated with this bit do not apply to the doorbell registers.
	007	rsvd	R	rsvd
	011:008	max_errlog_sz	R, IMPL	Indicates the maximum size of the error log for this function. The maximum error log size that is supported is 2**(max_errlog_sz + 23) bytes. Valid max_errlog_sz values range from 0 (8 MB) to 9 (4 GB); all other encodings are reserved.
	015:012	max_akey_sz	R, IMPL	Indicates the maximum size of any AKey table referenced by any context within this function. The maximum size that is supported is 2**(max_akey_sz+12) bytes. Values of max_akey_sz greater than 0x8 are reserved.
u16 max_cxt;	031:016		R, IMPL	The contexts supported by this function range from 0 to max_cxt. The maximum number of contexts supported by this function is max_cxt+1.
u32 opb_000_cap;	063:032		R, IMPL	Each bit in this field is set to 1 to indicate whether a certain descriptor operation-group is supported by the SDXI function. For a context to use a supported operation group, it must be made available in MMIO_CTL2.opb_000_avl and enabled in CXT_L1.opb_000_enb. See *"5.1 Descriptor Operations"* and *"Chapter 6, SDXI Descriptor and Operation Specification"* for more details.

Table 9–8: MMIO_VERSION[^1] (MMIO Offset: 0x0_0210)

Field	Bits	Subfield	Type & Reset	Description
u8 minor;	007:000		R, IMPL	Indicates Minor Version number of SDXI function implementation. 0x0 for the version of SDXI described in this document.
u8 rsvd_0;	015:008		R, 0x0	Reserved.
u8 major;	023:016		R, IMPL	Indicates Major Version number of SDXI function implementation. 0x1 for the version of SDXI described in this document.
u8 rsvd_1;	031:024		R, 0x0	Reserved.
u8 rsvd_2[4];	063:032		R, 0x0	Reserved.

9.2 Context and RKey Table Registers

Table 9–9: MMIO_CXT_L2[^1] (MMIO Offset: 0x1_0000^)

Field	Bits	Subfield	Type & Reset	Description
u64 lv02_ptr;	011:000	rsvd	R, 0x0	rsvd
	063:012	lv02_ptr	RW, 0x0	Pointer to the context level 2 table.

Table 9–10: MMIO_RKEY[^1] (MMIO Offset: 0x1_0100)

Field	Bits	Subfield	Type & Reset	Description
u64 ptr;	000	en	RW, 0x0	When set to 1, RKey functionality is enabled. Also, sz and ptr contain valid information.
	004:001	sz	RW, 0x0	Controls the size of the RKey table. This field is encoded the same as MMIO_CAP0.max_rkey_sz and must not exceed it in value. SNIASDXISpecification-v1.0 requires that software shall not program sz with values greater than 0x8.
	011:005	rsvd	R, 0x0	Rsvd
	063:012	ptr	RW, 0x0	Pointer to the start of the RKey table.

9.3 Error Logging Control and Status Registers

Table 9–11: MMIO_ERR_CTL[^1] (MMIO Offset:0x2_0000)

Field	Bits	Subfield	Type & Reset	Description
u64 intr_en;	000	en	RW, 0x0	1 = An interrupt is signaled when hardware transitions the MMIO_ERR_STS.sts bit from "0" to "1". When MSI or MSI-X is enabled, message 0 is signaled. 0 = No interrupt is generated
	063:001	rsvd	R, 0x0	rsvd

Table 9–12: MMIO_ERR_STS[^1] (MMIO Offset:0x2_0008)

Field	Bits	Subfield	Type & Reset	Description
u64 info;	000	sts	RW1C, 0x0	1 = An attempt to record an error in the log was made. See err bit for the success or failure of this operation. Once the SDXI function sets the sts bit, no new error interrupts, if enabled, will be generated until the bit is cleared (by software writing "1" to the bit). Additional errors may continue to be recorded in the log while the sts bit is "1". 0 = An error has not been recorded.
	001	ovf	RW1C, 0x0	1 = The error log has overflowed, and some error information was lost. MMIO_ERR_STS.err will be set on an overflow event. The ovf bit is a status bit and does not control operation of the error log. 0 = The error log has not overflowed.
	002	rsvd	R, 0x0	rsvd
	003	err	RW1C, 0x0	1 = An error occurred attempting to write the error log and some error information was lost. Further error logging and hardware modification of the MMIO_ERR_STS register is disabled until this bit is cleared (by software writing "1" to the bit). 0 = It has not.
	063:004	rsvd	R, 0x0	rsvd

Table 9–13: MMIO_ERR_CFG[^1] (MMIO Offset:0x2_0010)

Field	Bits	Subfield	Type & Reset	Description
u64 ptr;	000	en	RW, 0x0	When true, the error log mechanism is enabled and shall use the values of the MMIO_ERR_CFG.ptr and MMIO_ERR_CFG.sz fields.
	005:001	sz	RW, 0x0	This field controls the error log size. The size of the error log is 2**(sz + 12) bytes. The error log size must not exceed the max capability of the function as specified by MMIO_CAP1.max_errlog_sz.
	011:006	rsvd	R, 0x0	rsvd
	063:012	ptr	RW, 0x0	Pointer to the start of the error log.

Table 9–14: MMIO_ERR_WRT[^1] (MMIO Offset: 0x2_0020)

Field	Bits	Subfield	Type & Reset	Description
u64 index	063:000		RW,0x0	Description: Hardware error log write pointer. The function increments the write pointer when an error log entry is successfully written. The write pointer is not incremented when writing of an error log entry results in the MMIO_ERR_STS.err bit being set. The starting byte offset of the last error log entry written by the function is given by: - Let log_sz = 2**(MMIO_ERR_CFG.sz +12); - Let log_bs = MMIO_ERR_CFG & ~0xFFF; - Let index = MMIO_ERR_WRT – 1 - Then byte_offset = log_bs + ((index * 64) % log_sz); The error log is empty when MMIO_ERR_WRT == MMIO_ERR_RD. The write pointer value is expected to never wrap-around to 0. The memory buffer used by the error log may wrap around. Software may write MMIO_ERR_WRT only when error logging is disabled either through MMIO_ERR_CFG.en or MMIO_ERR_STS.err. Writes to this register at any other time result in undefined behavior. Software must always read this field with a 64-bit access regardless of the value the function returns in MMIO_CAP1.mmio64.

Table 9–15: MMIO_ERR_RD[^1] (MMIO Offset: 0x2_0028)

Field	Bits	Subfield	Type & Reset	Description
u64 index	063:000		RW,0x0	Description: Error log read pointer. This points to the starting byte offset of the first error log entry that has not been consumed (read) by software in this way: - Let log_sz = 2**(MMIO_ERR_CFG.sz +12); - Let log_bs = MMIO_ERR_CFG & ~0xFFF; - Then byte_offset = log_bs + ((MMIO_ERR_RD * 64) % log_sz); Software increments the read pointer as it consumes error log entries. Software must always write this field with a 64-bit access regardless of the value the function returns in MMIO_CAP1.mmio64.

9.4 MBOX Mailbox Registers

The PF/VF mailbox interface allows for low-bandwidth two-way communication under SR-IOV virtualization between privileged software on a PF and the software managing a VF. This may be used directly during runtime to pass control messages between privileged software and the VMs. It may also be used to bootstrap higher bandwidth forms of communication such as shared memory buffers, or even memory ring-buffers and interrupts that are written or generated by SDXI contexts

There is no mailbox communication directly between VFs. There is no mailbox communication directly between functions that belong to different SDXI SR-IOV devices even if they belong to the same SDXI Group. All mailbox registers are reserved in implementations that do not support SR-IOV. An implementation is only required to implement enough mailbox registers to support the number of VFs it implements.

Each mailbox is 16 bytes in size. There are separate transmit and receive mailboxes between each VF and the PF.

It is recommended that under virtualization, the Hypervisor take ownership of all SDXI PFs in order to facilitate communication between VFs under the same PF, and between SDXI VFs belonging to different PFs within the same SDXI function group.

VF_1 is the first virtual function under the PF. The PF will use a MMIO_MBS_CTL.tgt_vf value of 0 to set the MMIO_MBR_ST.rcv bit on VF_1. When VF_1 clears its rcv bit, the PF's MMIO_MBS_ST.ack bit 0 will become set. If the VF sends a message to the PF, the PF's MMIO_MBR_ST.rcv bit 0 will become set. Subsequent VFs, if present, will follow the same pattern. VF_N will use the tgt_vf value of N-1, ack bit N-1, and rcv bit N-1.

Table 9–16: MMIO_MB_CTL[^1] (MMIO Offset:0x3_0000)

Field	Bits	Subfield	Type & Reset	Description
u64 intr_en;	000	snd	RW, 0x0	Send message acknowledge interrupt enable. When 1, an interrupt is generated when any bit in MMIO_MBS_ST.ack is set to 1. When MSI or MSI-X is enabled, message 0 is signaled.
	001	rcv	RW, 0x0	Receive message interrupt enable. When 1, an interrupt is generated when any bit in MMIO_MBR_ST.rcv is set to 1. When MSI or MSI-X is enabled, message 0 is signaled.
	063:002	rsvd	R, 0x0	rsvd

Table 9–17: MMIO_MBS_CTL[^1] (MMIO Offset: 0x3_0010)

Field	Bits	Subfield	Type & Reset	Description
u16 msg;	000	msg	W	Writing this bit to 1 causes the target MMIO_MBR_ST.rcv bit to be set in the target function's mailbox. VF's always send to the PF mailbox.
	015:001	rsvd	0x0	rsvd
u16 tgt_vf;	031:016		W	Indicates the target VF number to be used by the msg field of this register. Shall be set to 0 for VF copies of this register.
u8 rsvd_0[4];	063:032		0x0	rsvd

Table 9–18: MMIO_MBS_ST[^1] (MMIO Offset: 0x3_1000 to 0x3_2FFF)

Field	Bits	Subfield	Type & Reset	Description
u64 ack;	063:000		RW1C	Message acknowledgment status. Indicates that the last message sent to the corresponding function was acknowledged by the receiver. VFs implements only bit 0 at offset 0x3_1000 representing a message received by the PF. The PF implements as many bits as there are VFs. This bit should be cleared prior to sending a new message to the target function.

Table 9–19: MMIO_MBR_ST[^1] (MMIO Offset: 0x3_3000 to 0x3_4FFF)

Field	Bits	Subfield	Type & Reset	Description
u64 rcv;	063:000		RW1C, 0x0	Receive message status. Indicates that rcv_data from the mailbox corresponding to the status bit is valid. VFs implements only bit 0 at offset 0x3_3000 representing a message received from the PF. The PF implements as many bits as there are VFs. Clearing this status bit returns an acknowledgement to the sender.

9.5 PF Mailbox Data Registers

Table 9–20: Send Mailbox Data Format

Field	Bits	Subfield	Type & Reset	Description
u8 msg[16];	127:000		RW	16 bytes of mailbox message data to be sent to a VF.

Table 9–21: Receive Mailbox Data Format

Field	Bits	Subfield	Type & Reset	Description
u8 msg[16];	127:000		R	16 bytes of mailbox message data received from a VF.

Table 9–22: MMIO Send Mailbox Registers

MMIO Addresses	MMIO Send Mailbox Range
0x06_0000 to 0x06_FFFF	MMIO_MBS[0000:0FFF]
0x08_0000 to 0x08_FFFF	MMIO_MBS[1000:1FFF]
0x0A_0000 to 0x0A_FFFF	MMIO_MBS[2000:2FFF]
0x0C_0000 to 0x0C_FFFF	MMIO_MBS[3000:3FFF]
0x0E_0000 to 0x0E_FFFF	MMIO_MBS[4000:4FFF]
0x10_0000 to 0x10_FFFF	MMIO_MBS[5000:5FFF]
0x12_0000 to 0x12_FFFF	MMIO_MBS[6000:6FFF]
0x14_0000 to 0x14_FFFF	MMIO_MBS[7000:7FFF]
0x16_0000 to 0x16_FFFF	MMIO_MBS[8000:8FFF]
0x18_0000 to 0x18_FFFF	MMIO_MBS[9000:9FFF]
0x1A_0000 to 0x1A_FFFF	MMIO_MBS[A000:AFFF]
0x1C_0000 to 0x1C_FFFF	MMIO_MBS[B000:BFFF]
0x1E_0000 to 0x1E_FFFF	MMIO_MBS[C000:CFFF]
0x20_0000 to 0x20_FFFF	MMIO_MBS[D000:DFFF]
0x22_0000 to 0x22_FFFF	MMIO_MBS[E000:EFFF]
0x24_0000 to 0x24_FFFF	MMIO_MBS[F000:FFFF]

Table 9–23: MMIO Receive Mailbox Registers

MMIO Addresses	MMIO Receive Mailbox Range
0x07_0000 to 0x07_FFFF	MMIO_MBR[0000:0FFF]
0x09_0000 to 0x09_FFFF	MMIO_MBR[1000:1FFF]
0x0B_0000 to 0x0B_FFFF	MMIO_MBR[2000:2FFF]
0x0D_0000 to 0x0D_FFFF	MMIO_MBR[3000:3FFF]
0x0F_0000 to 0x0F_FFFF	MMIO_MBR[4000:4FFF]
0x11_0000 to 0x11_FFFF	MMIO_MBR[5000:5FFF]
0x13_0000 to 0x13_FFFF	MMIO_MBR[6000:6FFF]
0x15_0000 to 0x15_FFFF	MMIO_MBR[7000:7FFF]
0x17_0000 to 0x17_FFFF	MMIO_MBR[8000:8FFF]
0x19_0000 to 0x19_FFFF	MMIO_MBR[9000:9FFF]
0x1B_0000 to 0x1B_FFFF	MMIO_MBR[A000:AFFF]
0x1D_0000 to 0x1D_FFFF	MMIO_MBR[B000:BFFF]
0x1F_0000 to 0x1F_FFFF	MMIO_MBR[C000:CFFF]
0x21_0000 to 0x21_FFFF	MMIO_MBR[D000:DFFF]
0x23_0000 to 0x23_FFFF	MMIO_MBR[E000:EFFF]
0x25_0000 to 0x25_FFFF	MMIO_MBR[F000:FFFF]

9.6 VF Mailbox Data Registers

Table 9–24: MMIO_MBS[^1] (MMIO Offset: 0x6_0000)

Field	Bits	Subfield	Type & Reset	Description
u8 msg[16];	127:000		RW	16 bytes of mailbox message data to be sent to the PF.

Table 9–25: MMIO_MBR[^1] (MMIO Offset: 0x7_0000)

Field	Bits	Subfield	Type & Reset	Description
u8 msg[16];	127:000		R	16 bytes of mailbox message data received from the PF.

9.7 Doorbell Sections and Registers

The SDXI function provides a per-context, MMIO-based Doorbell Section that is used by software to indicate that new work is available for that particular context. Software uses an interconnect-specific mechanism to map the start of this MMIO region (Doorbell_MMIO_START) with the SDXI function; for example, a PCI BAR. The MMIO region for Doorbell Sections is divided into doorbell-stride sized sections (based on MMIO_CAP0.db_stride). Each section, numbered from zero, corresponds to the context of the same number and is naturally aligned to a boundary of 2**(MMIO_CAP0.db_stride + 12) bytes. This spacing is provided to allow the doorbell region for a particular context to be mapped to a user process with MMU-based page protections. The size of the Doorbell MMIO region is dependent on the maximum supported doorbell stride and the maximum number of ring contexts supported:

Doorbell MMIO Region Size In Bytes =
(MMIO_CAP1.max_cxt + 1) * 2**(MMIO_CAP0.db_stride + 12)

The Doorbell MMIO region shall only be written; a read returns an undefined value.

The first 8-bytes of each section corresponds to the doorbell register for the matching context; the remaining bytes are reserved.

The doorbell register is used in the following way by software. After updating the context's Write_Index location and adding new descriptors to the context's ring, software performs an aligned 64-bit MMIO write of the Write_Index value, the doorbell_value, to its associated doorbell register address. The doorbell_value is evaluated per the rules specified in *"4.3.3, Doorbell Register and Context Signaling"*.

Figure 9-1: Doorbell MMIO Regions